

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):

Grupo de prácticas:

Fecha de entrega:

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: Si añadimos la cláusula `default(none)` estamos obligándonos a especificar el alcance de todas las variables usadas en la construcción (a excepción de índices de bucles con directivas `for` y variables `threadprivate`). Como en este caso no hemos especificado el alcance de `n`, el compilador nos devuelve un error informando de ello. La solución es especificar el ámbito de `n`, que será `shared` porque no tiene sentido hacer privado el número de elementos del vector (que será el mismo para todas las hebras).

CÓDIGO FUENTE: `shared-clauseModificado.c`

```
#include<stdio.h>
#ifdef _OPENMP
#include<omp.h>
#endif

int main(){
    int i,n = 7;
    int a[n] ;

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

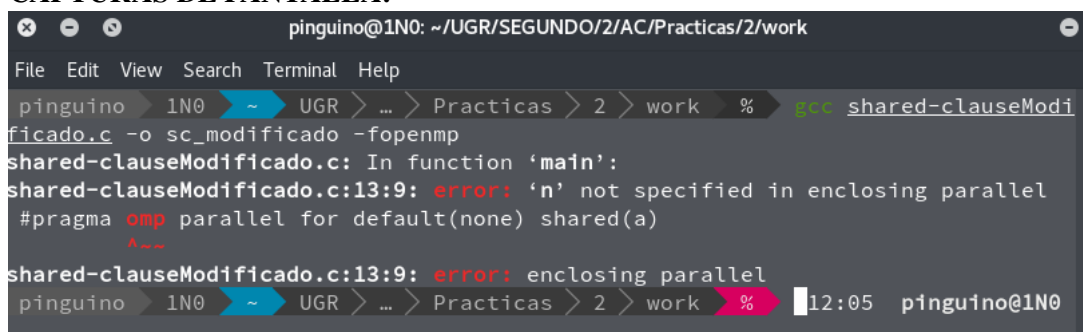
    #pragma omp parallel for default(none) shared(a,n)
        for(i=0;i<n;i++) a[i] += i;

    printf("Después de parallel for:\n");

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

    return(0);
}
```

CAPTURAS DE PANTALLA:



```
pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work
File Edit View Search Terminal Help
pinguino > 1N0 > ~ > UGR > ... > Practicas > 2 > work > % gcc shared-clauseModi
ficado.c -o sc_modificado -fopenmp
shared-clauseModificado.c: In function 'main':
shared-clauseModificado.c:13:9: error: 'n' not specified in enclosing parallel
#pragma omp parallel for default(none) shared(a)
      ^~~~
shared-clauseModificado.c:13:9: error: enclosing parallel
pinguino > 1N0 > ~ > UGR > ... > Practicas > 2 > work > % 12:05 pinguino@1N0
```

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:

Si se inicializa la variable `suma` fuera de la construcción `parallel` dicha inicialización no es válida porque *“El valor de entrada y de salida está indefinido aunque la variable esté declarada fuera de la construcción”* Es decir, si el valor de la variable `suma` (privada) se especifica fuera y no dentro del ámbito de la sección `parallel`, el valor que se le asigna no es tenido en cuenta y las variables contienen “basura”. En este caso el valor que toman es 0, y el resultado es “correcto”, pero probablemente se deba a que acabo de encender el ordenador y la memoria está libre en su mayoría, en otras circunstancias podría tomar otros valores. Si se inicializa el valor tanto dentro como fuera del ámbito de la sección `parallel`, la asignación que se tiene en cuenta es la de dentro. Si se asigna solo dentro del ámbito de la directiva `parallel`, la asignación se tiene en cuenta, y si no se inicializa en ninguno de los dos, la variable contiene para cada hebra “basura” aunque en este caso se da que también contiene 0 y el resultado es correcto, pero es una casualidad.

CÓDIGO FUENTE: `private-clauseModificado.c`

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

int main()
{
    int i, n = 7;
    int a[n], suma;

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

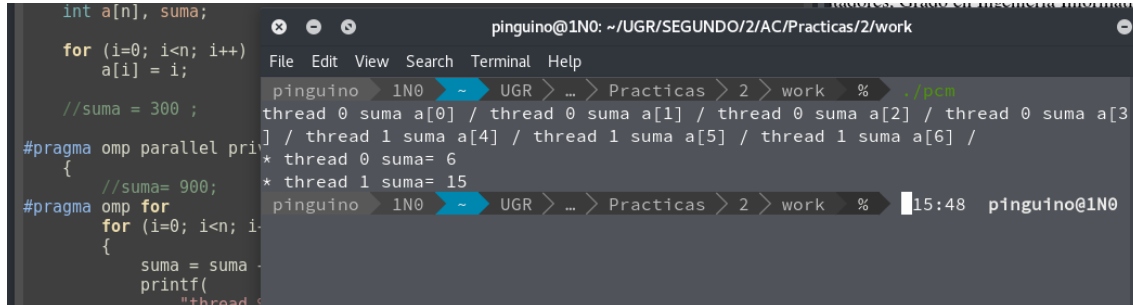
    //suma = 300 ;

    #pragma omp parallel private(suma)
    {
        suma= 900;
    #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");

    return 0;
}
```

CAPTURAS DE PANTALLA:



```

int a[n], suma;

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

//suma = 300 ;

#pragma omp parallel private(i)
{
    //suma= 900;
    #pragma omp for
    for (i=0; i<n; i++)
    {
        suma = suma + a[i];
        printf("thread %d suma %d\n", i, suma);
    }
}

```

pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work

File Edit View Search Terminal Help

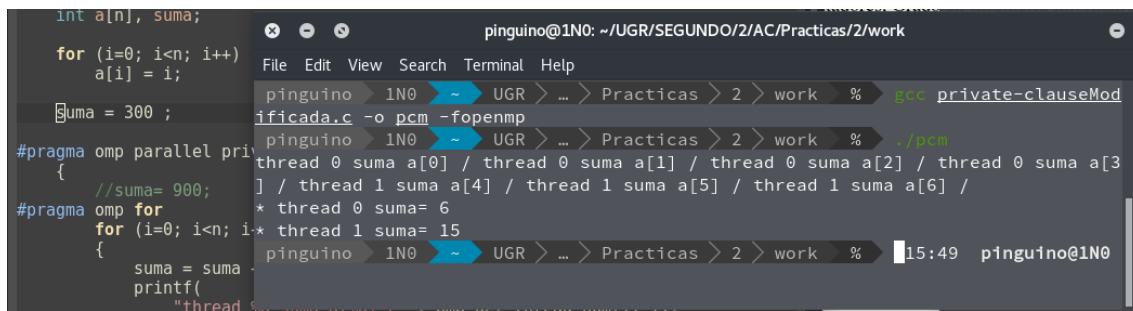
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./pcm

thread 0 suma a[0] / thread 0 suma a[1] / thread 0 suma a[2] / thread 0 suma a[3] / thread 1 suma a[4] / thread 1 suma a[5] / thread 1 suma a[6] /

* thread 0 suma= 6

* thread 1 suma= 15

pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % 15:48 pinguino@1N0



```

int a[n], suma;

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

//suma = 300 ;

#pragma omp parallel private(i)
{
    //suma= 900;
    #pragma omp for
    for (i=0; i<n; i++)
    {
        suma = suma + a[i];
        printf("thread %d suma %d\n", i, suma);
    }
}

```

pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work

File Edit View Search Terminal Help

pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % gcc private-clauseMod

ificada.c -o pcm -fopenmp

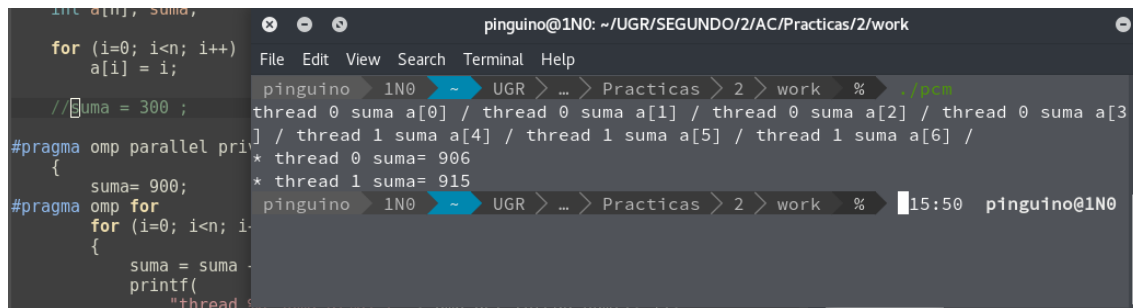
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./pcm

thread 0 suma a[0] / thread 0 suma a[1] / thread 0 suma a[2] / thread 0 suma a[3] / thread 1 suma a[4] / thread 1 suma a[5] / thread 1 suma a[6] /

* thread 0 suma= 6

* thread 1 suma= 15

pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % 15:49 pinguino@1N0



```

int a[n], suma;

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

//suma = 300 ;

#pragma omp parallel private(i)
{
    suma= 900;
    #pragma omp for
    for (i=0; i<n; i++)
    {
        suma = suma + a[i];
        printf("thread %d suma %d\n", i, suma);
    }
}

```

pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work

File Edit View Search Terminal Help

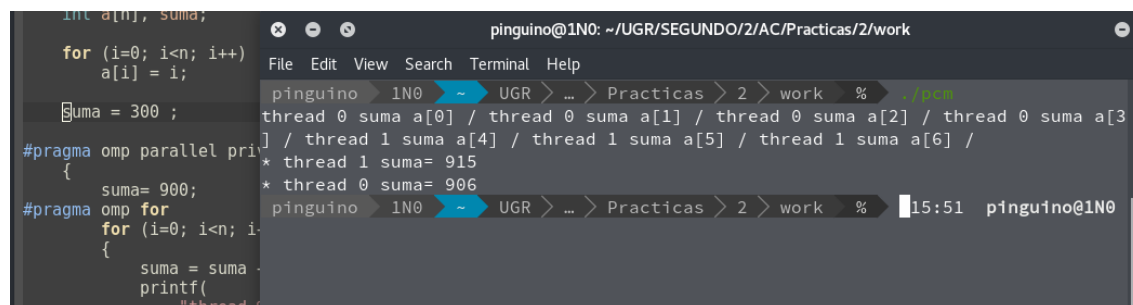
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./pcm

thread 0 suma a[0] / thread 0 suma a[1] / thread 0 suma a[2] / thread 0 suma a[3] / thread 1 suma a[4] / thread 1 suma a[5] / thread 1 suma a[6] /

* thread 0 suma= 906

* thread 1 suma= 915

pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % 15:50 pinguino@1N0



```

int a[n], suma;

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

//suma = 300 ;

#pragma omp parallel private(i)
{
    suma= 900;
    #pragma omp for
    for (i=0; i<n; i++)
    {
        suma = suma + a[i];
        printf("thread %d suma %d\n", i, suma);
    }
}

```

pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work

File Edit View Search Terminal Help

pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./pcm

thread 0 suma a[0] / thread 0 suma a[1] / thread 0 suma a[2] / thread 0 suma a[3] / thread 1 suma a[4] / thread 1 suma a[5] / thread 1 suma a[6] /

* thread 0 suma= 915

* thread 1 suma= 906

pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % 15:51 pinguino@1N0

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

RESPUESTA: Que la suma obtenida es igual para todos los threads y puede dar cualquier valor entre 0 y la suma total de los elementos del vector (21). Esto se debe a que cada thread ejecuta la asignación `suma = 0` y, el valor final es, dado que todas las hebras van a sumar a la misma variable y pueden darse condiciones de carrera que provoquen que alguna suma no se efectúe, y dado que solo se almacenarán las sumas efectuadas después de la última asignación “`suma = 0`” (porque las demás se sobrescribirán), lo más probable es que no se efectúen la mayoría de las sumas. Sin embargo, podría darse el caso (y en efecto a mí se me dio en la primera ejecución y he subido una captura de pantalla) de que se hicieran todas las instrucciones “`suma=0`” al principio y luego no se diera la circunstancia de que dos sumas se realizaran concurrentemente de forma que una tapara a la otra y al final el resultado fuera 21 (que es el valor de sumar los números desde 0 hasta 7)

CÓDIGO FUENTE: `private-clauseModificado3.c`

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

int main()
{
    int i, n = 7;
    int a[n], suma;

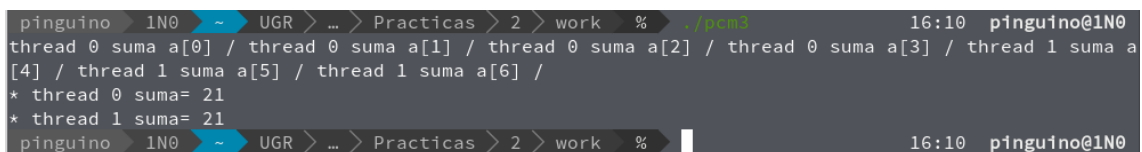
    for (i=0; i<n; i++)
        a[i] = i;

    #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* thread %d suma= %d", omp_get_thread_num(), suma);
    }

    printf("\n");

    return 0;
}
```

CAPTURAS DE PANTALLA:



```
pinguino ~ 1N0 UGR > ... > Practicas > 2 > work % ./pwn3 16:10 pinguino@1N0
thread 0 suma a[0] / thread 0 suma a[1] / thread 0 suma a[2] / thread 0 suma a[3] / thread 1 suma a
[4] / thread 1 suma a[5] / thread 1 suma a[6] /
* thread 0 suma= 21
* thread 1 suma= 21
pinguino ~ 1N0 UGR > ... > Practicas > 2 > work % 16:10 pinguino@1N0
```

```

Applications - Terminal
pinguino@IN0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work

libgomp: Invalid value for environment variable OMP_NUM_THREADS
thread 1 suma a[1] / thread 3 suma a[3] / thread 4 suma a[4] / thread 2 suma a[2] / thread 0 suma a[0] / thread 5 suma a[5] / thread 6 suma a[6] /
* thread 5 suma= 5
* thread 1 suma= 5
* thread 0 suma= 5
* thread 3 suma= 5
* thread 2 suma= 5
* thread 4 suma= 5
* thread 6 suma= 5
* thread 7 suma= 5
pinguino@IN0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work % ./practica2

libgomp: Invalid value for environment variable OMP_NUM_THREADS
thread 0 suma a[0] / thread 2 suma a[2] / thread 3 suma a[3] / thread 1 suma a[1] / thread 5 suma a[5] / thread 6 suma a[6] / thread 4 suma a[4] /
* thread 6 suma= 6
* thread 7 suma= 6
* thread 1 suma= 6
* thread 3 suma= 6
* thread 2 suma= 6
* thread 4 suma= 6
* thread 0 suma= 6
* thread 5 suma= 6
pinguino@IN0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work % ./practica2

libgomp: Invalid value for environment variable OMP_NUM_THREADS
thread 1 suma a[1] / thread 6 suma a[6] / thread 5 suma a[5] / thread 0 suma a[0] / thread 4 suma a[4] / thread 2 suma a[2] / thread 3 suma a[3] /
* thread 5 suma= 3
* thread 4 suma= 3
* thread 1 suma= 3
* thread 7 suma= 3
* thread 2 suma= 3
* thread 6 suma= 3
* thread 3 suma= 3
pinguino@IN0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work % ./practica2

libgomp: Invalid value for environment variable OMP_NUM_THREADS
thread 3 suma a[3] / thread 5 suma a[5] / thread 6 suma a[6] / thread 4 suma a[4] / thread 1 suma a[1] / thread 0 suma a[0] / thread 2 suma a[2] /
* thread 6 suma= 2
* thread 1 suma= 2
* thread 2 suma= 2
* thread 7 suma= 2
* thread 3 suma= 2
* thread 5 suma= 2
* thread 4 suma= 2
pinguino@IN0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work %

```

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:

Si, se imprime siempre el 6 porque la directiva `lastprivate` provoca que la variable tenga fuera de la construcción `for` el valor que tendría si el bucle se ejecutara secuencialmente, y es independiente del orden en que se ejecuten las hebras.

CAPTURAS DE PANTALLA:

```

Applications - Terminal
pinguino@IN0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work % ./practica2

libgomp: Invalid value for environment variable OMP_NUM_THREADS
thread 6 suma a[6] suma=6
thread 0 suma a[0] suma=0
thread 3 suma a[3] suma=3
thread 2 suma a[2] suma=2
thread 1 suma a[1] suma=1
thread 4 suma a[4] suma=4
thread 5 suma a[5] suma=5

Fuera de la construcción parallel suma=6
pinguino@IN0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work %

```

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: Si se elimina la cláusula `copyprivate(a)` en la directiva `single` se está haciendo que cada tread tenga su propia variable `a` (privada pues ha sido declarada dentro de la construcción `parallel`) que no ha sido inicializada (luego contiene basura), luego en el vector aparecen valores no determinados (en la captura de pantalla se ven algunas posiciones con valor 0 pero es pura coincidencia).

CÓDIGO FUENTE: copyprivate-clauseModificado.c

```

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

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

    return 0;
}

```

CAPTURAS DE PANTALLA:

```

pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work
File Edit View Search Terminal Help

libgomp: Invalid value for environment variable OMP_NUM_THREADS

Introduce valor de inicialización a: 5

Single ejecutada por el thread 6
Después de la región parallel:
b[0] = 4196645  b[1] = 4196645  b[2] = 0      b[3] = 0      b[4] = 0      b[5] = 0      b[6]
] = 0      b[7] = 5      b[8] = 0
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./copyprivate 17:21 pinguino@1N0

libgomp: Invalid value for environment variable OMP_NUM_THREADS

Introduce valor de inicialización a: 1

Single ejecutada por el thread 6
Después de la región parallel:
b[0] = 4196645  b[1] = 4196645  b[2] = 0      b[3] = 0      b[4] = 0      b[5] = 0      b[6]
] = 0      b[7] = 1      b[8] = 0
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % 17:21 pinguino@1N0

```

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

RESPUESTA: Si cambiamos la inicialización de la variable `suma` se obtiene el mismo valor sumándole el valor al que se inicialice, en este caso, 10. Es decir, se obtiene la suma realizada en el bucle sumada al valor con que se inicializa `suma` (10)

CÓDIGO FUENTE: `reduction-clauseModificado.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, 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:

The screenshot shows a terminal window titled 'pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work'. The user runs a program with various arguments, and the output shows the final value of 'suma' after a parallel reduction. The results are as follows:

Initial Value of suma	Final Value of suma (after parallel reduction)
20	20
3	13
1	10
2	11
3	13
4	16
5	20

The terminal output shows the command `./reductionModificado` followed by the initial value, and the output `Tras 'parallel' suma=X` where X is the final result.

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: Como no se usan directivas de trabajo compartido la manera que se me ocurre de realizar el mismo trabajo es utilizar una variable privada auxiliar “sumalocal” que se utiliza en el bucle `for` y que luego se acumula en la variable `suma` utilizando la directiva `atomic` para evitar condiciones de carrera.

CÓDIGO FUENTE: `reduction-clauseModificado7.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, n=20, a[n], suma=0, sumalocal = 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 firstprivate(sumalocal) shared(suma)
    {
        #pragma omp for
        for (i=0; i<n; i++)    sumalocal += a[i];
        #pragma omp atomic
        suma += sumalocal ;
    }
    printf("Tras 'parallel' suma=%d\n", suma);
}
```

CAPTURAS DE PANTALLA:

```
pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work
File Edit View Search Terminal Help
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % gcc reductionClauseMo
dificado7.c -o rm7 -fopenmp
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./rm7 5
Tras 'parallel' suma=10
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./rm7 10
Tras 'parallel' suma=45
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % gcc reduction-clause.
c -o rm -fopenmp
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./rm 5
Tras 'parallel' suma=10
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % ./rm 10
Tras 'parallel' suma=45
```


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, v2, 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

```
/* pmv-secuencial.c
Producto de una matriz cuadrada M por un vector v1
Para compilar usar (-lrt: real time library)
gcc -O2 pmv-secuencial.c -o pmv -lrt
*/

#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()

//Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de los ...
//tres defines siguientes puede estar descomentado):
//#define VECTOR_LOCAL // descomentar para que los vectores sean variables ...
// locales (si se supera el tamaño de la pila se ...
// generará el error "Violación de Segmento")
#define VECTOR_GLOBAL // descomentar para que los vectores sean variables ...
// globales (su longitud no estará limitada por el ...
// tamaño de la pila del programa)
//#define VECTOR_DYNAMIC // descomentar para que los vectores sean variables ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifdef VECTOR_GLOBAL
#define MAX 1000 //2^25
double M[MAX][MAX], v1[MAX], v2[MAX];
#endif

int main(int argc, char** argv){
    int i;
    struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecución
    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2){
        printf("Error: Falta el número de filas y columnas.\n");
        exit(-1);
    }
    unsigned int f = atoi(argv[1]); // Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)
#ifdef VECTOR_LOCAL
    double M[f][f], v1[f], v2[f]; // Tamaño variable local en tiempo de ejecución ...
    // disponible en C a partir de actualización C99
#endif
#ifdef VECTOR_GLOBAL
    if (f>MAX) f=MAX;
#endif

#ifdef VECTOR_DYNAMIC
    double **M = (double **)malloc(f*sizeof(double*)) ;
    for( i = 0 ; i < f ; i++) M[i] = (double*)malloc(f * sizeof(double)) ;
    double *v1, *v2 ;
    v1 = (double*) malloc(f*sizeof(double)); // malloc necesita el tamaño en bytes
    v2 = (double*) malloc(f*sizeof(double)); //si no hay espacio suficiente malloc devuelve NULL
    if ( (v1==NULL) || (v2==NULL) || (M==NULL) ){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }
}
#endif

//Inicializar vector
for(i=0; i<f; i++){
    v1[i] = 1.0;
}
```

```

//Inicializar matriz
int j ;
for(i=0 ; i < f ; i++){
    for(j = 0 ; j < f ; j++){
        M[i][j] = 2.0;
    }

    clock_gettime(CLOCK_REALTIME,&cgt1);
    //Calcular multiplicación.
    for(i=0; i<f; i++){
        v2[i] = 0 ;
        for(j=0 ; j<f ; j++){
            v2[i] += M[i][j]*v1[j] ;
        }
    }

    clock_gettime(CLOCK_REALTIME,&cgt2);
    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
        (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));

    printf("Tiempo(seg.): %11.9f\t / filas: %u\t / columnas: %u\t",
        ncgt,f,f) ;
    for(i = 0 ; i < f ; i++){
        printf("\nv1[%d]= %11.3f",i,v1[i]);
    }
    for(i=0 ; i < f ; i++)
        printf("\nM[1][%d]=%11.3f",i,M[1][i]);
    for(i=0 ; i < f ; i++)
        printf("\nv2[%d]= %11.3f",i,v2[i]);
#ifdef VECTOR_DYNAMIC
    for(i = 0 ; i < f ; i++)
        free(M[i]) ;
    free(M) ;
    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
#endif
    return 0;
}

```

CAPTURAS DE PANTALLA:

```

pinguino ~ UGR > ... > Practicas > 2 > work % gcc pmv-secuencial.c
-o pmv-secuencial -lrt
pinguino ~ UGR > ... > Practicas > 2 > work % ./pmv-secuencial 5
Tiempo(seg.): 0.000002263 / filas: 5 / columnas: 5
v1[0]= 1.000
v1[1]= 1.000
v1[2]= 1.000
v1[3]= 1.000
v1[4]= 1.000
M[1][0]= 2.000
M[1][1]= 2.000
M[1][2]= 2.000
M[1][3]= 2.000
M[1][4]= 2.000
v2[0]= 10.000
v2[1]= 10.000
v2[2]= 10.000
v2[3]= 10.000
v2[4]= 10.000%
pinguino ~ UGR > ... > Practicas > 2 > work % 22:46 pinguino@1N0

```

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

```
/* pmv-secuencial.c
   Producto de una matriz cuadrada M por un vector v1
   Para compilar usar (-lrt: real time library)
   gcc -O2 pmv-secuencial.c -o pmv -lrt
*/

#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()

//Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de los ...
//tres defines siguientes puede estar descomentado):
//#define VECTOR_LOCAL // descomentar para que los vectores sean variables ...
// locales (si se supera el tamaño de la pila se ...
// generará el error "Violación de Segmento")
//define VECTOR_GLOBAL// descomentar para que los vectores sean variables ...
// globales (su longitud no estará limitada por el ...
// tamaño de la pila del programa)
#define VECTOR_DYNAMIC// descomentar para que los vectores sean variables ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifdef VECTOR_LOCAL
#define MAX 1000 //2^25
double M[MAX][MAX], v1[MAX], v2[MAX];
#endif

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

    int i;

    struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecución

    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2){
        printf("Error: Falta el número de filas y columnas.\n");
        exit(-1);
    }
    unsigned int f = atoi(argv[1]);// Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)
#ifdef VECTOR_LOCAL
    double M[f][f], v1[f], v2[f]; // Tamaño variable local en tiempo de ejecución ...
    // disponible en C a partir de actualización C99
#endif
#ifdef VECTOR_GLOBAL
    if (f>MAX) f=MAX;
#endif
#ifdef VECTOR_DYNAMIC
    double **M = (double**)malloc(f*sizeof(double*)) ;
    for( i = 0 ; i < f ; i++) M[i] = (double*)malloc(f * sizeof(double)) ;
    double *v1, *v2 ;
```

```

v1 = (double*) malloc(sizeof(double)); // malloc necesita el tamaño en bytes
v2 = (double*) malloc(sizeof(double)); //si no hay espacio suficiente malloc devuelve NULL
if ( (v1==NULL) || (v2==NULL) || (M==NULL) ){
    printf("Error en la reserva de espacio para los vectores\n");
    exit(-2);
}
#endif

//Inicializar vector
#pragma omp for
for(i=0; i<f; i++){
    v1[i] = 1.0;
}
//Inicializar matriz
int j;
#pragma omp for
for(i=0 ; i < f ; i++)
    for(j = 0 ; j < f ; j++){
        M[i][j] = 2.0;
    }
int h;
int VECES = 123;
for(h = 0 ; h < VECES ; h++){

#pragma omp parallel for
for(i = 0 ; i < f ; i++)
    v2[i] = 0 ;

    clock_gettime(CLOCK_REALTIME,&cgt1);
    //Calcular multiplicación. Descomposición por filas
#pragma omp parallelprivate(j)
    {
        #pragma omp for
        for(i=0; i<f; i++){
            for(j=0 ; j<f ; j++){
                v2[i] += M[i][j]*v1[j] ;
            }
        }
    }
    clock_gettime(CLOCK_REALTIME,&cgt2);
    ncgt += (double) (cgt2.tv_sec-cgt1.tv_sec)+
        (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));

    int k ;

    printf("\n%d\n",h) ;

    for(k = 0 ; k < f ; k++){
        printf("\nv1[%d]= %11.3f",k,v1[k]);
    }

    for(k=0 ; k < f ; k++)
        printf("\nM[1][%d]=%11.3f",k,M[1][k]);
    for(k=0 ; k < f ; k++)
        printf("\nv2[%d]= %11.3f",k,v2[k]);

}
ncgt = ncgt/VECES ;
printf("\nTiempo(seg.): %11.9f / filas: %u / columnas: %u",
    ncgt,f,f) ;
#ifdef VECTOR_DYNAMIC
for(i = 0 ; i < f ; i++)
    free(M[i]) ;
free(M) ;
free(v1); // libera el espacio reservado para v1
free(v2); // libera el espacio reservado para v2
#endif
return 0;
}

```

CÓDIGO FUENTE: pmv-openMP-b.c

```

/* pmv-secuencial.c
Producto de una matriz cuadrada M por un vector v1
Para compilar usar (-lrt: real time library)
gcc -O2 pmv-secuencial.c -o pmv -lrt
*/

#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()

//Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de los ...
//tres defines siguientes puede estar descomentado):
//#define VECTOR_LOCAL // descomentar para que los vectores sean variables ...
// locales (si se supera el tamaño de la pila se ...
// generará el error "Violación de Segmento")
//#define VECTOR_GLOBAL // descomentar para que los vectores sean variables ...
// globales (su longitud no estará limitada por el ...

```

```

// tamaño de la pila del programa)
#define VECTOR_DYNAMIC// descomentar para que los vectores sean variables ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifdef VECTOR_GLOBAL
#define MAX 1000 //2^25
double M[MAX][MAX], v1[MAX], v2[MAX], vP[MAX];
#endif

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

    int i;

    struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecución

    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2){
        printf("Error: Falta el número de filas y columnas.\n");
        exit(-1);
    }
    unsigned int f = atoi(argv[1]);// Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)
#ifdef VECTOR_LOCAL
    double M[f][f], v1[f], v2[f], vP[f]; // Tamaño variable local en tiempo de ejecución ...
    // disponible en C a partir de actualización C99
#endif
#ifdef VECTOR_GLOBAL
    if (f>MAX) f=MAX;
#endif
#ifdef VECTOR_DYNAMIC
    double **M = (double **)malloc(*sizeof(double*));
    for( i = 0 ; i < f ; i++) M[i] = (double*)malloc(f * sizeof(double));
    double *v1, *v2, *vP;
    v1 = (double*) malloc(*sizeof(double)); // malloc necesita el tamaño en bytes
    v2 = (double*) malloc(*sizeof(double)); //si no hay espacio suficiente malloc devuelve NULL
    if ( (v1==NULL) || (v2==NULL) || (M==NULL)){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }
#endif

    //Inicializar vector
    #pragma omp for
    for(i=0; i<f; i++){
        v1[i] = 1.0;
    }
    //Inicializar matriz
    int j;
    for(i=0 ; i < f ; i++){
    #pragma omp for
        for(j = 0 ; j < f ; j++){
            M[i][j] = 2.0;
        }
    }

    #pragma omp parallel for
    for(int i = 0 ; i < f ; i++){
        v2[i] = 0;
    }

    int h;
    int VECES = 123 ;
    for(h = 0 ; h < VECES ; h++){

        clock_gettime(CLOCK_REALTIME,&cgt1);
        //Calcular multiplicación. Descomposición por columnas
        #pragma omp parallel private(i,vP) shared(v2)
        {
            #ifdef VECTOR_DYNAMIC
                vP = (double*) malloc(*sizeof(double)); //si no hay espacio suficiente malloc devuelve NULL
            #endif
            for(i=0; i<f; i++){
                #pragma omp for
                for(j=0 ; j<f ; j++){
                    vP[i] += M[i][j]*v1[j];
                }
            }

            for(int i = 0 ; i < f ; i++)
                #pragma omp atomic
                v2[i] += vP[i];
            #ifdef VECTOR_DYNAMIC
                free(vP);
            #endif
        }

        clock_gettime(CLOCK_REALTIME,&cgt2);
        ncgt+=(double) (cgt2.tv_sec-cgt1.tv_sec)+
            (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));

        int k;

        printf("\n%d\n",h);

        for(k = 0 ; k < f ; k++){
            printf("\nv1[%d]= %11.3f",k,v1[k]);
        }
    }
}

```

```

for(k=0 ; k < f ; k++)
    printf("\nM[1][%d]=%11.3f",k,M[1][k]);
for(k=0 ; k < f ; k++)
    printf("\nv2[%d]= %11.3f",k,v2[k]);
}
ncgt = ncgt/VECES ;
printf("\nTiempo(seg.): %11.9f / filas: %u / columnas: %u",
ncgt,f,f) ;
#ifdef VECTOR_DYNAMIC
for(i = 0 ; i < f ; i++)
    free(M[i]) ;
free(M) ;
free(v1); // libera el espacio reservado para v1
free(v2); // libera el espacio reservado para v2
#endif
return 0;
}

```

RESPUESTA:**CAPTURAS DE PANTALLA:**

The screenshots show the following terminal output:

```

pinguino@1N0: ~/UGR/SEGUNDO/2/AC/Practicas/2/work
pinguino ~ IN0 UGR > ... > Practicas > 2 > work % gcc pmv-secuencial.c -o pmvS -fopenmp -lrt
pinguino ~ IN0 UGR > ... > Practicas > 2 > work % gcc pmv-OpenMP-a.c -o pmvA -fopenmp -lrt
pinguino ~ IN0 UGR > ... > Practicas > 2 > work % gcc pmv-OpenMP-b.c -o pmvB -fopenmp -lrt
pinguino ~ IN0 UGR > ... > Practicas > 2 > work % 19:53 pinguino@1N0

v2[12342]= 24690.000
v2[12343]= 24690.000
v2[12344]= 24690.000
Tiempo(seg.): 0.497291073 / filas: 12345 / columnas: 12345
pinguino ~ IN0 UGR > ... > Practicas > 2 > work % ./pmvS 12345

v2[12342]= 24690.000
v2[12343]= 24690.000
v2[12344]= 24690.000
Tiempo(seg.): 0.491058220 / filas: 12345 / columnas: 12345
pinguino ~ IN0 UGR > ... > Practicas > 2 > work % ./pmvA 12345

v2[12342]= 187909204.000
v2[12343]= 187909204.000
v2[12344]= 187909204.000
Tiempo(seg.): 0.260442934 / filas: 12345 / columnas: 12345
pinguino ~ IN0 UGR > ... > Practicas > 2 > work % ./pmvB 12345

```

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

```

/* pmv-secuencial.c
Producto de una matriz cuadrada M por un vector v1
Para compilar usar (-lrt: real time library)
gcc -O2 pmv-secuencial.c -o pmv -lrt
*/

#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()

//Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de los ...
//tres defines siguientes puede estar descomentado):
//#define VECTOR_LOCAL // descomentar para que los vectores sean variables ...
// locales (si se supera el tamaño de la pila se ...
// generará el error "Violación de Segmento")
//#define VECTOR_GLOBAL // descomentar para que los vectores sean variables ...
// globales (su longitud no estará limitada por el ...
// tamaño de la pila del programa)
#define VECTOR_DYNAMIC // descomentar para que los vectores sean variables ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifndef VECTOR_GLOBAL
#define MAX 1000 //2^25
double M[MAX][MAX], v1[MAX], v2[MAX], vP[MAX];
#endif

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

    int i;

    struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecución

    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2){
        printf("Error: Falta el número de filas y columnas.\n");
        exit(-1);
    }
    unsigned int f = atoi(argv[1]); // Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)
    #ifdef VECTOR_LOCAL
        double M[f][f], v1[f], v2[f]; // Tamaño variable local en tiempo de ejecución ...
        // disponible en C a partir de actualización C99
    #endif
    #ifdef VECTOR_GLOBAL
        if (f>MAX) f=MAX;
    #endif
    #ifdef VECTOR_DYNAMIC
        double **M = (double**)malloc(f*sizeof(double*)) ;
        for( i = 0 ; i < f ; i++) M[i] = (double*)malloc(f * sizeof(double)) ;
        double *v1, *v2 ;
        v1 = (double*) malloc(f*sizeof(double)); // malloc necesita el tamaño en bytes
        v2 = (double*) malloc(f*sizeof(double)); //si no hay espacio suficiente malloc devuelve NULL
        if ( (v1==NULL) || (v2==NULL) || (M==NULL)){
            printf("Error en la reserva de espacio para los vectores\n");
            exit(-2);
        }
    #endif

    //Inicializar vector
    #pragma omp for
    for(i=0; i<f; i++){
        v1[i] = 1.0;
    }
    //Inicializar matriz
    int j ;
    for(i=0 ; i < f ; i++)
    #pragma omp for
        for(j = 0 ; j < f ; j++){
            M[i][j] = 2.0;
        }

    #pragma omp parallel for
    for(int i = 0 ; i < f ; i++){
        v2[i] = 0 ;
    }

    int h;
    int VECES = 123 ;
    for(h = 0 ; h < VECES ; h++){
        clock_gettime(CLOCK_REALTIME,&cgt1);
        //Calcular multiplicación. Descomposición por columnas
        #pragma omp parallel private(i) shared(v2)
        {
            for(i=0; i<f; i++){
                #pragma omp for reduction(+:v2[i])
                for(j=0 ; j<f ; j++){
                    v2[i] += M[i][j]*v1[j] ;
                }
            }
        }

        clock_gettime(CLOCK_REALTIME,&cgt2);
        ncgt+=(double) (cgt2.tv_sec-cgt1.tv_sec)+
            (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
    }
}

```

```

int k ;

printf("\n%d\n",h) ;

for(k = 0 ; k < f ; k++){
    printf("\nv1[%d]= %11.3f",k,v1[k]);
}

for(k=0 ; k < f ; k++)
    printf("\nM[1][%d]=%11.3f",k,M[1][k]);
for(k=0 ; k < f ; k++)
    printf("\nv2[%d]= %11.3f",k,v2[k]);
}
ncgt = ncgt/VECES ;
printf("\nTiempo(seg.): %11.9f\t / filas: %u\t / columnas: %u\t",
    ncgt,f,f) ;
#ifdef VECTOR_DYNAMIC
    for(i = 0 ; i < f ; i++)
        free(M[i]) ;
    free(M) ;
    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
#endif
return 0;
}

```

RESPUESTA: El uso de la cláusula reduction es “equivalente” a utilizar una variable privada para acumular los cálculos como hice en el ejercicio anterior.

CAPTURAS DE PANTALLA:

```

v2[7]=      2952.000
v2[8]=      2952.000
v2[9]=      2952.000
v2[10]=     2952.000
v2[11]=     2952.000
Tiempo(seg.): 0.000212731      / filas: 12      / columnas: 12 %
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % gcc pmv-OpenMP-reduct
ion.c -o pmvREDUCTION -fopenmp -lrt
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % 20:16 pinguino@1N0

```

```

pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % gcc pmv-OpenMP-reduct
ion.c -o pmvREDUCTION -fopenmp -lrt
pmv-OpenMP-reduction.c: In function 'main':
pmv-OpenMP-reduction.c:84:26: error: user defined reduction not found for 'v2'
#pragma omp for reduction(+:v2)
                          ^
pinguino 1N0 ~ UGR > ... > Practicas > 2 > work % 20:12 pinguino@1N0

```

Fallo: he intentado reducir el vector entero, lo que no tiene sentido ninguno. Hay que hacer `reduction(+:v2[i])` para cada `i`.

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):

COMENTARIOS SOBRE LOS RESULTADOS:

Para comprobar cual de las implementaciones es mejor he medido el tiempo que tardan para un tamaño grande (dado que el código está implementado para devolver el tiempo medio de 100 ejecuciones, estas medidas deberían ser fiables).

Tiempos obtenidos para un tamaño 12345:

Secuencial: 0.504872889
 OpenMP-a: 0.172282166
 OpenMP-b: 0.167361843
 OpenMP-Reduction: 0.169658529

Aunque en mi opinión la opción que paraleliza las filas parece más apropiada, al medir los tiempos compruebo que (aunque si bien es cierto, no hay mucha diferencia) las dos últimas opciones dan tiempos mejores. Entre esas dos últimas, la que mejor tiempo ha devuelto ha sido la opción que no utiliza la cláusula reduction, pero la diferencia no es mucha. Decido ejecutar otra vez todas las versiones para asegurarme de que los resultados son buenos. (Esta vez he ejecutado con un tamaño menor, 123, pero he aumentado el número de ejecuciones a 1000 con la variable VECES)

Tiempos obtenidos para un tamaño 123:

Secuencial: 0.000053723
 OpenMP-a: 0.000084277
 OpenMP-b: 0.000109173
 OpenMP-Reduction: 0.000234336

La idea no ha sido muy buena, para tamaños pequeños el tiempo empleado en la paralelización hace que sean mejores los tiempos obtenidos secuencialmente.

Definitivamente he decidido utilizar el código con la cláusula reduction porque aunque el tiempo obtenido era peor que el del segundo código, creo que puede obtener tiempos iguales o mejores que este (ya que son “equivalentes”).

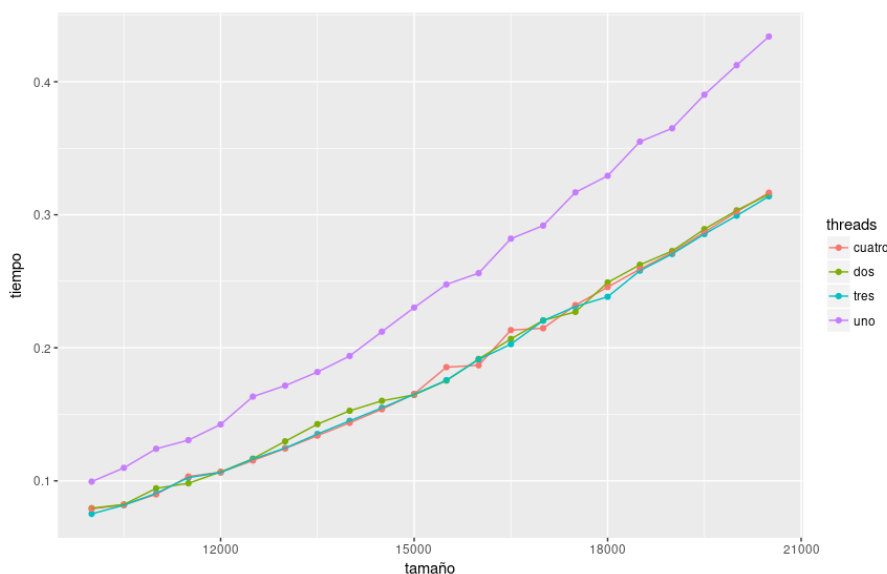


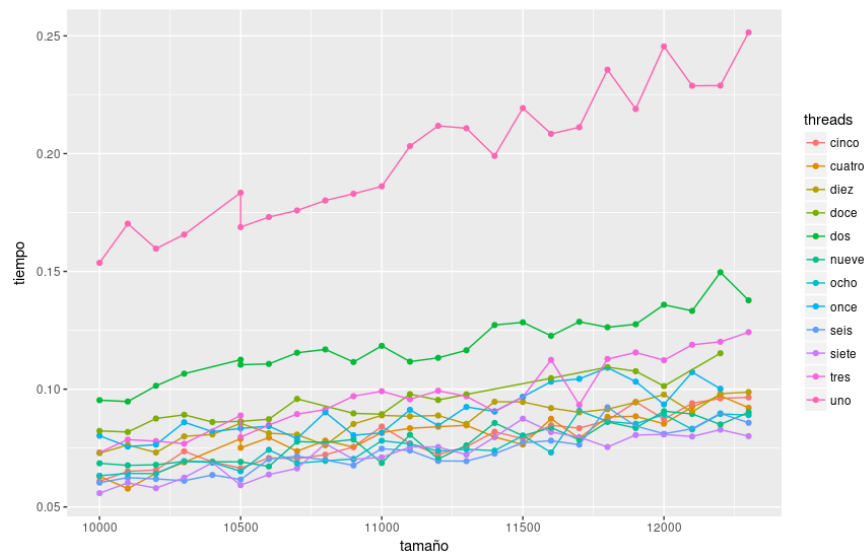
Illustration 1: Tiempos en local

Local: (msi cx61)

Threads Tamaño	uno	dos	tres	cuatro
10000	0.098954	0.073622	0.078892	0.073945
10100	0.1052	0.075203	0.075265	0.075462
10200	0.102708	0.081563	0.077539	0.081938
10300	0.10513	0.078709	0.078675	0.078563
10400	0.111028	0.082478	0.08406	0.079943
10500	0.109128	0.081737	0.081493	0.086361
10600	0.115541	0.083197	0.08311	0.083375
10700	0.114442	0.084394	0.089324	0.089818
10800	0.115713	0.086951	0.086871	0.086137
10900	0.121175	0.092895	0.088436	0.087761
11000	0.119706	0.089023	0.08921	0.098966
11100	0.122175	0.090399	0.091598	0.091005
11200	0.12416	0.100946	0.092674	0.092518
11300	0.126235	0.093732	0.098626	0.10397
11400	0.128879	0.095657	0.09765	0.096119
11500	0.135971	0.100417	0.097651	0.097634
11600	0.133212	0.098981	0.106794	0.101386
11700	0.135396	0.100474	0.101072	0.101019
11800	0.144301	0.102936	0.103205	0.102456
11900	0.141163	0.113094	0.104484	0.104262
12000	0.142456	0.105709	0.110437	0.110511
12100	0.145424	0.107118	0.108526	0.112356
12200	0.15122	0.108707	0.109569	0.109474
12300	0.153175	0.115184	0.111924	0.111235

Atc-grid:

threads Tamaño	uno	dos	tres	cuatro	cinco	seis	siete	ocho	nueve	diez	once	doce
10000	0.153634	0.09532	0.073122	0.06298	0.060849	0.060316	0.055845	0.063246	0.068535	0.07287	0.080224	0.082281
10500	0.183418	0.112486	0.08884	0.07895	0.065115	0.06243	0.060234	0.064246	0.067646	0.076281	0.075624	0.081811
10100	0.170276	0.094737	0.078566	0.057773	0.065644	0.061889	0.057963	0.064138	0.067895	0.07313	0.076442	0.087537
10200	0.159678	0.101424	0.077958	0.064259	0.073632	0.061127	0.062422	0.06942	0.069319	0.079964	0.08596	0.089126
10300	0.165652	0.106573	0.076836	0.068977	0.069123	0.063572	0.068838	0.068875	0.069173	0.080786	0.08196	0.086051
10500	0.168788	0.110378	0.079614	0.075109	0.066454	0.061633	0.059229	0.065128	0.069146	0.085494	0.083349	0.086333
10600	0.173072	0.110716	0.084788	0.079414	0.070802	0.070316	0.063761	0.07424	0.067185	0.081275	0.084286	0.087242
10700	0.175869	0.115466	0.089404	0.073707	0.070522	0.071488	0.066359	0.068569	0.077757	0.08071	0.079009	0.095858
10800	0.180112	0.116874	0.09133	0.078141	0.072167	0.069923	0.076844	0.069558	0.077365	0.076183	0.090187	0.934321
10900	0.182955	0.111527	0.097009	0.075418	0.075475	0.06763	0.070216	0.070382	0.078646	0.085228	0.080445	0.08972
11000	0.186126	0.118384	0.099121	0.081559	0.084146	0.074724	0.071105	0.078108	0.068667	0.088865	0.081436	0.089378
11100	0.203196	0.111698	0.095718	0.0834	0.076936	0.073947	0.075488	0.076778	0.080632	0.088478	0.091209	0.097875
11200	0.211807	0.113301	0.099309	0.084046	0.072279	0.069525	0.075441	0.073705	0.070383	0.088814	0.084524	0.095407
11300	0.210758	0.116521	0.096966	0.084661	0.07554	0.069441	0.072213	0.074496	0.076132	0.085253	0.092446	0.097777
11400	0.199045	0.127237	0.090533	0.0797	0.081919	0.072626	0.079677	0.073903	0.085648	0.094661	0.090551	0.098988
11500	0.219334	0.128371	0.096622	0.076485	0.078972	0.07723	0.087465	0.080281	0.080233	0.09457	0.096784	0.100023
11600	0.208406	0.122642	0.112422	0.087433	0.084676	0.078131	0.081778	0.07313	0.083657	0.091965	0.103187	0.104655
11700	0.211187	0.128608	0.093338	0.079152	0.083437	0.076419	0.079628	0.090807	0.078526	0.090146	0.104381	0.108923
11800	0.235637	0.126286	0.112814	0.088245	0.086767	0.092283	0.075438	0.086316	0.086086	0.091497	0.109118	0.109346
11900	0.21895	0.127551	0.115569	0.08847	0.094584	0.084372	0.080544	0.085281	0.083524	0.094443	0.103198	0.107615
12000	0.245516	0.135841	0.112277	0.085313	0.087258	0.081056	0.080762	0.08932	0.090643	0.097721	0.093509	0.115255
12100	0.228812	0.13323	0.118847	0.092479	0.094006	0.083206	0.079907	0.08296	0.089469	0.090228	0.107216	0.115534
12200	0.228898	0.149613	0.120107	0.097223	0.096078	0.089723	0.082797	0.089481	0.085013	0.098076	0.100164	0.115643
12300	0.251448	0.137724	0.124167	0.092099	0.096523	0.085728	0.080058	0.089001	0.090199	0.098742	0.100265	0.115661



A continuación he calculado la ganancia para dos tamaños: 12300 y 11100.

Calculando la ganancia como $\frac{T_s}{T_p(p)}$ dónde T_s es el tiempo con un thread (secuencial) y $T_p(p)$ es el tiempo en paralelo con p procesadores. Obtengo:

Local:

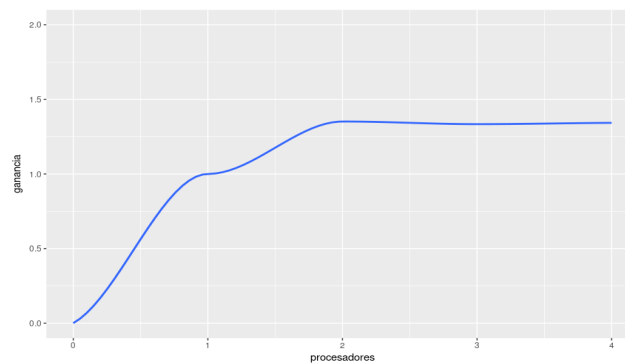
11100:

threads	1	2	3	4
tiempo	0.122175	0.090399	0.091598	0.091005
ganancia	1	1.35150831314506	1.33381733225616	1.34250865337069

12300:

threads	1	2	3	4
tiempo	0.153175	0.115184	0.111924	0.111235
ganancia	1	1.32982879566606	1.36856259604732	1.37703960084506

De aquí puedo generar una gráfica para ver la escalabilidad del programa en mi sistema:



Por su parte, en `atc_grid`, los resultados obtenidos para 11100 y 12300 en cuanto a ganancia son:

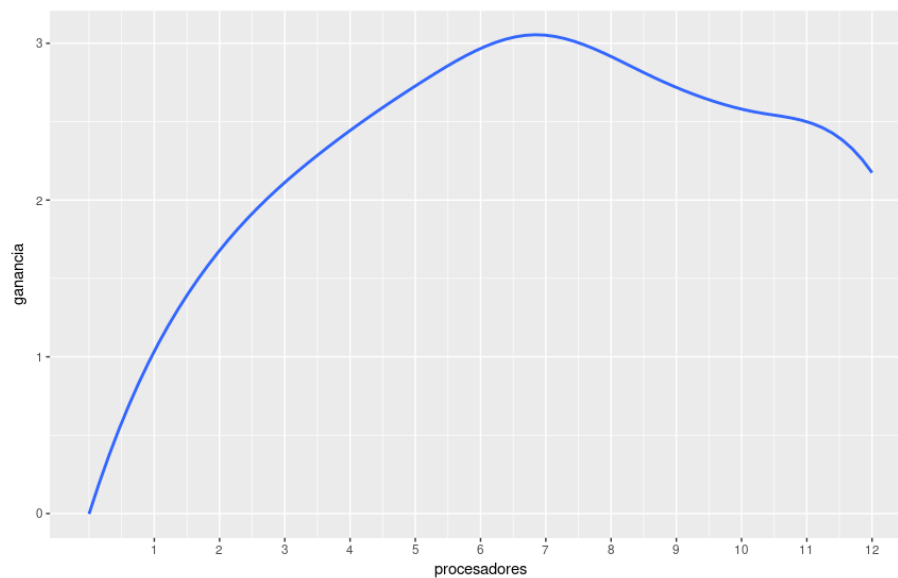
11100:

threads	1	2	3	4	5	6	7	8	9	10	11	12
tiempo	0.203196	0.111698	0.095718	0.0834	0.076936	0.073947	0.075488	0.076778	0.080632	0.088478	0.091209	0.097875
ganancia	1	1.8191555	1.8191552	2.4364028	2.6411042	2.7478599	2.6917655	2.6465393	2.5200416	2.2965708	2.2278064	2.0760766

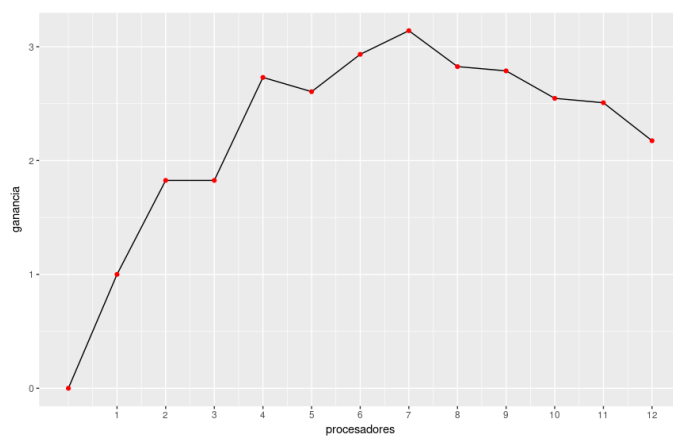
12300:

threads	1	2	3	4	5	6	7	8	9	10	11	12
tiempo	0.251448	0.137724	0.124167	0.092099	0.096523	0.085728	0.080058	0.089001	0.090199	0.098742	0.100265	0.115661
ganancia	1	3338852	3338852	1023355	6185676	0548712	0339504	8284626	4615018	6072188	3926595	2491332

Que gráficamente queda:



*Nota: he graficado un ajuste con una función polinómica de grado 9 para ilustrar como varía la ganancia en función de los procesadores, la gráfica real es la siguiente:



El error en la “aproximación” es mucho mayor que en la gráfica de los datos locales, pero ilustra bien cómo varía la ganancia con el número de procesadores y el punto (7 procesadores) en el que el overhead (sobrecarga) produce pérdidas en la ganancia.

Esto sirve para ilustrar que la escalabilidad de un programa está limitada por el paralelismo de este en cuanto al número de procesadores/threads que se aprovechan y que, llegados a un punto (dos threads en mi ordenador, 7 en el atcgrid) el aumento de procesadores no mejora el tiempo de ejecución e incluso lo empeora (debido al Overhead producido por la comunicación/sincronización de threads, la creación y terminación de los mismos, la falta de equilibrio, las colisiones en el acceso a memoria, cálculos o funciones añadidas que no estaban en la versión secuencial y son necesarias en la versión paralela, etc..)