

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): Daniel Bolaños Martínez

Grupo de prácticas: A1

Fecha de entrega: 20/04/17

Fecha evaluación en clase: 21/04/17

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)`? **(a)** (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:

Da un error de compilación debido a que poniendo la cláusula `default(none)` cambia el ámbito por defecto de las variables a “ninguno”, por lo que debemos especificar (en este caso en el `shared`) todas las variables que se utilicen dentro del código en paralelo. Por ello, debemos declarar `n`.

CÓDIGO FUENTE: `shared-clauseModificado.c`

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

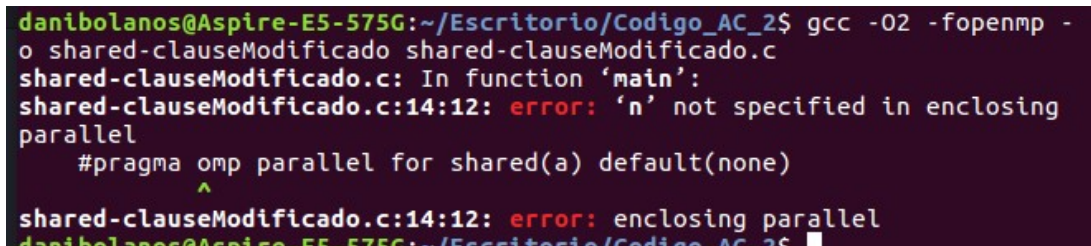
int main()
{
    int i, n = 7;
    int a[n];

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

    #pragma omp parallel for shared(a,n) default(none)
    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:


```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp -
o shared-clauseModificado shared-clauseModificado.c
shared-clauseModificado.c: In function 'main':
shared-clauseModificado.c:14:12: error: 'n' not specified in enclosing
parallel
    #pragma omp parallel for shared(a) default(none)
                   ^
shared-clauseModificado.c:14:12: error: enclosing parallel
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$

```

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 inicializamos la variable `suma` fuera de la región `parallel`, esta tendrá un valor indeterminado, para que el valor sea el correcto, debemos inicializarlo dentro de la región o usando la cláusula `firstprivate`.

Inicializando dentro y fuera de la región `parallel`, los resultados finales serán los correctos puesto que la inicialización dentro del `parallel`, sobrescribe a la de fuera.

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=2;

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

    #pragma omp parallel private(suma)
    {
        suma=1;
        #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:

private-clauseModificado2: variable suma inicializada a 2 fuera del `parallel`.

private-clauseModificado: variable suma inicializada dentro (a 1) y fuera (a 2) del `parallel`.

```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./private-ClauseMo
dificado2
thread 3 suma a[6] / thread 2 suma a[4] / thread 2 suma a[5] / thread 1
suma a[2] / thread 1 suma a[3] / thread 0 suma a[0] / thread 0 suma a[
1] /
* thread 3 suma= 4196566
* thread 2 suma= 4196569
* thread 1 suma= 4196565
* thread 0 suma= 5
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./private-ClauseMo
dificado
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 2 suma a[4] / thread 2 suma a[
5] /
* thread 0 suma= 2
* thread 1 suma= 6
* thread 3 suma= 7
* thread 2 suma= 10

```

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

RESPUESTA:

La variable `suma` se comparte entre todas las hebras al estar fuera de la región `parallel` y por tanto, todos los resultados son siempre el mismo, ya que cada hebra, trabaja sobre la misma variable y sobrescriben el resultado de la anterior.

CÓDIGO FUENTE: `private-clauseModificado3.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=2;

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

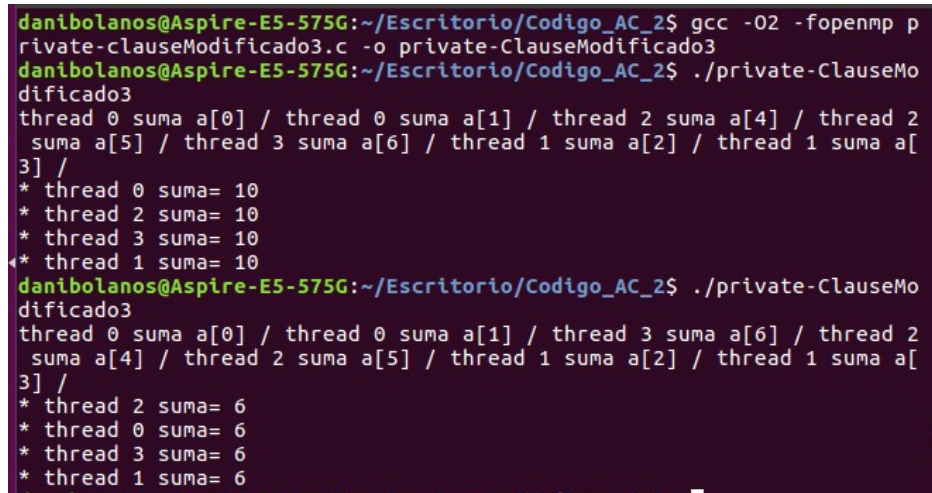
    #pragma omp parallel
    {
        suma=1;
        #pragma omp for
        for (i=0; i<n; i++)
        {
            suma = suma + a[i];

```

```

        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:


```

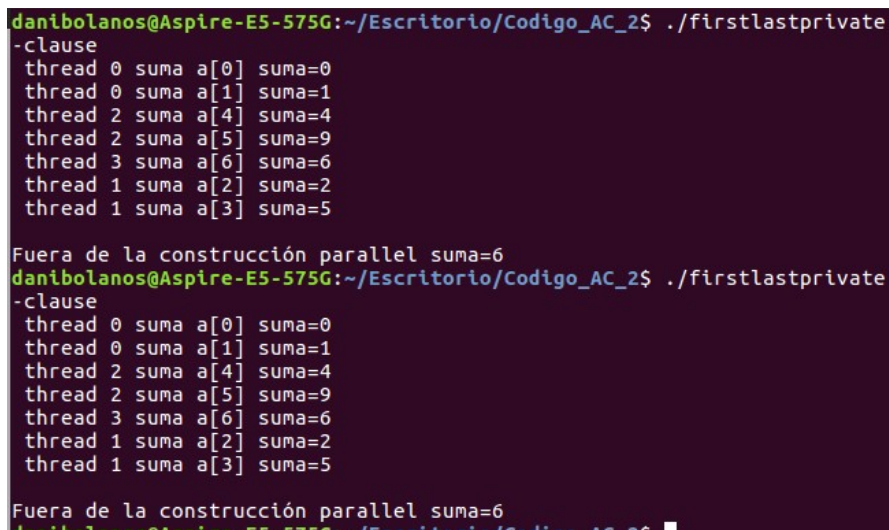
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp private-clauseModificado3.c -o private-ClauseModificado3
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./private-ClauseModificado3
thread 0 suma a[0] / thread 0 suma a[1] / thread 2 suma a[4] / thread 2 suma a[5] / thread 3 suma a[6] / thread 1 suma a[2] / thread 1 suma a[3] /
* thread 0 suma= 10
* thread 2 suma= 10
* thread 3 suma= 10
* thread 1 suma= 10
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./private-ClauseModificado3
thread 0 suma a[0] / thread 0 suma a[1] / thread 3 suma a[6] / thread 2 suma a[4] / thread 2 suma a[5] / thread 1 suma a[2] / thread 1 suma a[3] /
* thread 2 suma= 6
* thread 0 suma= 6
* thread 3 suma= 6
* thread 1 suma= 6

```

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 ponemos las dos cláusulas conjuntas `firstprivate` y `lastprivate`, el resultado de la suma fuera de `parallel` siempre da 6, debido a que `lastprivate` asigna a la variable `suma` el último valor (en una ejecución secuencial) de las variables de la lista, en este caso el de la última hebra creada; como estamos usando 4 hebras, la 3, cuya suma siempre vale 6.

CAPTURAS DE PANTALLA:


```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./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
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./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
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$

```

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 programa no funciona correctamente porque la variable `a` introducida por teclado no se copia (por difusión) al resto de threads y por tanto el vector no se inicializa completamente a ese valor como debería hacerse para el correcto funcionamiento del programa.

Algunos valores quedan bien inicializados, ya que lo hace la hebra que ejecutó el `single` y el resto quedan con valores indeterminados

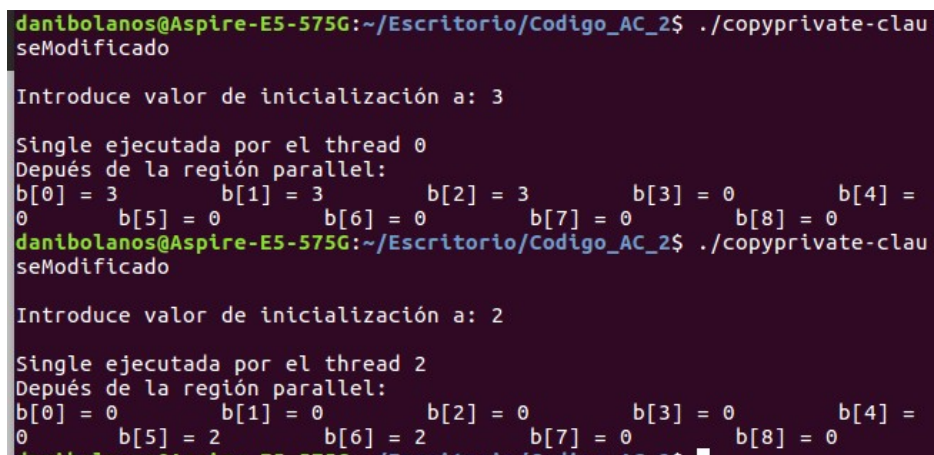
CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

```
#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");
}
```

CAPTURAS DE PANTALLA:



```
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./copyprivate-clauseModificado
Introduce valor de inicialización a: 3
Single ejecutada por el thread 0
Después de la región parallel:
b[0] = 3      b[1] = 3      b[2] = 3      b[3] = 0      b[4] = 0
0      b[5] = 0      b[6] = 0      b[7] = 0      b[8] = 0
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./copyprivate-clauseModificado
Introduce valor de inicialización a: 2
Single ejecutada por el thread 2
Después de la región parallel:
b[0] = 0      b[1] = 0      b[2] = 0      b[3] = 0      b[4] = 0
0      b[5] = 2      b[6] = 2      b[7] = 0      b[8] = 0
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$
```

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

RESPUESTA:

La suma resultante se incrementa también en 10 unidades.

Esto ocurre porque la cláusula `reduction` funciona de la siguiente manera, primero suma todos los valores privados de las iteraciones del bucle y finalmente, se los suma a la variable inicial (`suma`). Por tanto se incrementa en el valor en la que ésta esté inicializada.

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 const **argv) {
    int i, n = 20, a[n], suma = 10;

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

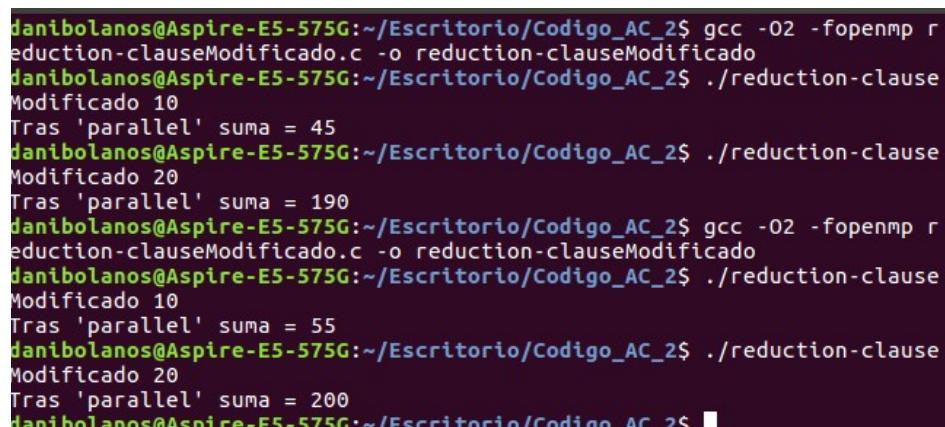
    n = atoi(argv[1]); if(n > 20){n = 20; printf("n = %d\n",n);}

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

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

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

CAPTURAS DE PANTALLA:



```
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp r
eduction-clauseModificado.c -o reduction-clauseModificado
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./reduction-clause
Modificado 10
Tras 'parallel' suma = 45
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./reduction-clause
Modificado 20
Tras 'parallel' suma = 190
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp r
eduction-clauseModificado.c -o reduction-clauseModificado
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./reduction-clause
Modificado 10
Tras 'parallel' suma = 55
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./reduction-clause
Modificado 20
Tras 'parallel' suma = 200
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$
```


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:

Lo que tenemos que hacer es incluir la directiva `#pragma omp atomic` (funciona como `critical`, de manera más eficiente) dentro del bucle `for` de la región paralela, para evitar que cada hebra pueda acceder a la variable `suma` de forma simultánea y se produzcan errores en el resultado.

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 const **argv) {
    int i, n = 20, a[n], suma = 0;

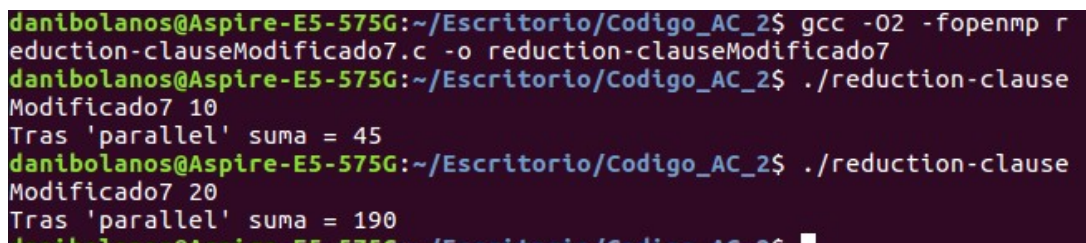
    if(argc < 2){
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }

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

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

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

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

CAPTURAS DE PANTALLA:


```
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp r
eduction-clauseModificado7.c -o reduction-clauseModificado7
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./reduction-clause
Modificado7 10
Tras 'parallel' suma = 45
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./reduction-clause
Modificado7 20
Tras 'parallel' suma = 190
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$
```

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):

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

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

```
/* pmv-secuencial.c
Multiplicación de matriz por vector: v2 = M1*v1
Para compilar usar (-lrt: real time library):
gcc -O2 -fopenmp pmv-secuencial.c -o pmv-secuencial -lrt

Para ejecutar use: pmv-secuencial longitud
*/

#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()
#include <omp.h>

#define PRINTF_ALL // comentar para quitar el printf ...
// que imprime todos los componentes

//Sólo puede estar definida uno de los dos constantes PMV_ (sólo uno
de los ...
//un defines siguientes puede estar descomentado):
//#define PMV_GLOBAL // descomentar para que los vector/matriz sean
variables ...
// globales (su longitud no estará
limitada por el ...
// tamaño de la pila del programa)
#define PMV_DYNAMIC // descomentar para que los vector/matriz sean
variables ...
// dinámicas (memoria reutilizable
durante la ejecución)
#ifdef PMV_GLOBAL
#define MAX 16350 // aprox 2^14
double M1[MAX][MAX], v1[MAX], v2[MAX];
#endif
```



```

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

    int i, k;

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

    //Leer argumento de entrada (no de componentes del vector/matriz)
    if (argc<2){
        printf("Faltan no componentes del vector/matriz\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)

    #ifdef PMV_GLOBAL
    if (N>MAX) N=MAX;
    #endif
    #ifdef PMV_DYNAMIC
    double **M1, *v1, *v2;
    M1 = (double**) malloc(N*sizeof(double*)); // malloc necesita el
    tamaño en bytes
    v1 = (double*) malloc(N*sizeof(double)); //si no hay espacio
    suficiente malloc devuelve NULL
    v2 = (double*) malloc(N*sizeof(double));
    if ( (M1==NULL) || (v1==NULL) || (v2==NULL) ){
        printf("Error en la reserva de espacio para los
    vectores/matriz\n");
        exit(-2);
    }
    for(i=0; i<N;i++){
        M1[i] = (double*) malloc(N*sizeof(double));
        if ( M1[i]==NULL ){
            printf("Error en la reserva de espacio para la matriz\n");
            exit(-2);
        }
    }
    #endif

    //Inicializar Matriz y vector
    for(i=0; i<N; i++){
        v1[i] = N*0.1-i*0.1; //los valores dependen de N
        v2[i]=0;
    }
    for(i=0; i<N; i++){
        for(k=0; k<N; k++)
            M1[i][k] = N*0.1-i*0.1-k*0.1; //los valores dependen de N
    }

    cgt1 = omp_get_wtime();
    //Calcular producto Matriz*vector
    for(i=0; i<N; i++){
        for(k=0; k<N; k++)
            v2[i] += M1[i][k] * v1[k];
    }
}

```

```

    cgt2 = omp_get_wtime();
    ncgt= cgt2-cgt1;

    //Imprimir resultado de la suma y el tiempo de ejecución
    #ifdef PRINTF_ALL
    printf("Tiempo(seg.):%11.9f\t / Tamaño Matriz-Vector:
    %u\n",ncgt,N);
    for(i=0; i<N; i++)
        printf("V2[%d]=%8.6f\n", i, v2[i]);

    #else
    printf("Tiempo(seg.):%11.9f\t / Tamaño Matriz-Vector:%u\t/
    V2[0]=%8.6f / / V2[%d]=%8.6f /\n", ncgt,N, v2[0], N-1, v2[N-1]);
    #endif

    #ifdef PMV_DYNAMIC
    for(i=0; i<N;i++)
        free(M1[i]);
    free(M1); // libera el espacio reservado para M1
    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
    #endif
    return 0;
}

```

CAPTURAS DE PANTALLA:

```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp p
mv-secuencial.c -o pmv-secuencialGlobal -lrt
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp p
mv-secuencial.c -o pmv-secuencialDinamico -lrt
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-secuencialGl
obal 8
Tiempo(seg.):0.000001540          / Tamaño Matriz-Vector:8
V2[0]=2.040000
V2[1]=1.680000
V2[2]=1.320000
V2[3]=0.960000
V2[4]=0.600000
V2[5]=0.240000
V2[6]=-0.120000
V2[7]=-0.480000
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-secuencialDi
namico 8
Tiempo(seg.):0.000001313          / Tamaño Matriz-Vector:8
V2[0]=2.040000
V2[1]=1.680000
V2[2]=1.320000
V2[3]=0.960000
V2[4]=0.600000
V2[5]=0.240000
V2[6]=-0.120000
V2[7]=-0.480000

```

```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-secuencialG
obal 11
Tiempo(seg.):0.000002146          / Tamaño Matriz-Vector:11
V2[0]=5.060000
V2[1]=4.400000
V2[2]=3.740000
V2[3]=3.080000
V2[4]=2.420000
V2[5]=1.760000
V2[6]=1.100000
V2[7]=0.440000
V2[8]=-0.220000
V2[9]=-0.880000
V2[10]=-1.540000
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-secuencialD
namico 11
Tiempo(seg.):0.000001766          / Tamaño Matriz-Vector:11
V2[0]=5.060000
V2[1]=4.400000
V2[2]=3.740000
V2[3]=3.080000
V2[4]=2.420000
V2[5]=1.760000
V2[6]=1.100000
V2[7]=0.440000
V2[8]=-0.220000
V2[9]=-0.880000
V2[10]=-1.540000

```

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
Multiplicación de matriz por vector: v2 = M1*v1
Para compilar usar (-lrt: real time library):
gcc -O2 -fopenmp pmv-OpenMP-a.c -o pmv-OpenMP-a -lrt

Para ejecutar use: pmv-OpenMP-a longitud

```

```

*/

#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()
#include <omp.h>

#define PRINTF_ALL // comentar para quitar el printf ...
                    // que imprime todos los componentes

//Sólo puede estar definida uno de los dos constantes PMV_ (sólo uno
de los ...
//un defines siguientes puede estar descomentado):
//#define PMV_GLOBAL // descomentar para que los vector/matriz sean
variables ...
                    // globales (su longitud no estará
limitada por el ...
                    // tamaño de la pila del programa)
#define PMV_DYNAMIC // descomentar para que los vector/matriz sean
variables ...
                    // dinámicas (memoria reutilizable
durante la ejecución)
#ifdef PMV_GLOBAL
#define MAX 16350 // aprox 2^14
double M1[MAX][MAX], v1[MAX], v2[MAX];
#endif

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

    int i, k;
    double suma_fila;

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

    omp_set_num_threads(2); //para cambiar el n.º threads usadas

    //Leer argumento de entrada (no de componentes del vector/matriz)
    if (argc<2){
        printf("Faltan no componentes del vector/matriz\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)

    #ifdef PMV_GLOBAL
    if (N>MAX) N=MAX;
    #endif
    #ifdef PMV_DYNAMIC
    double **M1, *v1, *v2;
    M1 = (double**) malloc(N*sizeof(double*)); // malloc necesita el
tamaño en bytes
    v1 = (double*) malloc(N*sizeof(double)); //si no hay espacio

```

```

suficiente malloc devuelve NULL
v2 = (double*) malloc(N*sizeof(double));
if ( (M1==NULL) || (v1==NULL) || (v2==NULL) ){
    printf("Error en la reserva de espacio para los
vectores/matriz\n");
    exit(-2);
}
for(i=0; i<N;i++){
    M1[i] = (double*) malloc(N*sizeof(double));
    if ( M1[i]==NULL ){
        printf("Error en la reserva de espacio para la matriz\n");
        exit(-2);
    }
}
#endif

//Inicializar matriz y vectores
#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<N; i++){
        v1[i] = N*0.1-i*0.1; //los valores dependen de N
        v2[i]=0;
    }

    #pragma omp for private(k)
    for(i=0; i<N; i++){
        for(k=0; k<N; k++)
            M1[i][k] = N*0.1-i*0.1-k*0.1; //los valores dependen
de N
    }
}

cgt1 = omp_get_wtime();

//Calcular producto Matriz*vector
//Acumula el valor de la multiplicación de una fila de la matriz
por el vector (v1)
//en la variable suma_filas y luego se la asigna al vector
resultado (v2)

#pragma omp parallel for private(suma_filas, k)
for(i=0; i<N; i++){
    suma_filas = 0;
    for(k=0; k<N; k++)
        suma_filas += M1[i][k]*v1[k];
    v2[i] = suma_filas;
}

cgt2 = omp_get_wtime();

ncgt= cgt2-cgt1;

//Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
printf("Tiempo(seg.):%11.9f\t / Tamaño Matriz-Vector:

```

```

%u\n", ncgt, N);
    for(i=0; i<N; i++)
        printf("V2[%d]=%8.6f\n", i, v2[i]);

    #else
        printf("Tiempo(seg.):%11.9f\t / Tamaño Matriz-Vector:%u\t/
V2[0]=%8.6f / / V2[%d]=%8.6f /\n", ncgt, N, v2[0], N-1, v2[N-1]);
    #endif

    #ifdef PMV_DYNAMIC
        for(i=0; i<N;i++)
            free(M1[i]);
        free(M1); // libera el espacio reservado para M1
        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
Multiplicación de matriz por vector: v2 = M1*v1
Para compilar usar (-lrt: real time library):
gcc -O2 -fopenmp pmv-OpenMP-b.c -o pmv-OpenMP-b -lrt

Para ejecutar use: pmv-OpenMP-b longitud
*/

#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()
#include <omp.h>

#define PRINTF_ALL // comentar para quitar el printf ...
                    // que imprime todos los componentes

//Sólo puede estar definida uno de los dos constantes PMV_ (sólo uno
de los ...
//un defines siguientes puede estar descomentado):
//#define PMV_GLOBAL // descomentar para que los vector/matriz sean
variables ...
                    // globales (su longitud no estará
limitada por el ...
                    // tamaño de la pila del programa)
#define PMV_DYNAMIC // descomentar para que los vector/matriz sean
variables ...
                    // dinámicas (memoria reutilizable
durante la ejecución)
#ifdef PMV_GLOBAL
#define MAX 16350 // aprox 2^14
double M1[MAX][MAX], v1[MAX], v2[MAX];
#endif

```



```

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

    int i, k;
    double suma_colum;

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

    omp_set_num_threads(2); //para cambiar el n.º threads usadas

    //Leer argumento de entrada (no de componentes del vector/matriz)
    if (argc<2){
        printf("Faltan no componentes del vector/matriz\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)

    #ifdef PMV_GLOBAL
    if (N>MAX) N=MAX;
    #endif
    #ifdef PMV_DYNAMIC
    double **M1, *v1, *v2;
    M1 = (double**) malloc(N*sizeof(double*)); // malloc necesita el
    tamaño en bytes
    v1 = (double*) malloc(N*sizeof(double)); //si no hay espacio
    suficiente malloc devuelve NULL
    v2 = (double*) malloc(N*sizeof(double));
    if ( (M1==NULL) || (v1==NULL) || (v2==NULL) ){
        printf("Error en la reserva de espacio para los
    vectores/matriz\n");
        exit(-2);
    }
    for(i=0; i<N;i++){
        M1[i] = (double*) malloc(N*sizeof(double));
        if ( M1[i]==NULL ){
            printf("Error en la reserva de espacio para la matriz\n");
            exit(-2);
        }
    }
    #endif

    //Inicializar matriz y vectores
    #pragma omp parallel
    {
        #pragma omp for
        for(i=0; i<N; i++){
            v1[i] = N*0.1-i*0.1; //los valores dependen de N
            v2[i]=0;
        }

        #pragma omp for private(k)
        for(i=0; i<N; i++){
            for(k=0; k<N; k++)
                M1[i][k] = N*0.1-i*0.1-k*0.1; //los valores dependen

```

```

de N
    }
}

cgt1 = omp_get_wtime();

//Calcular producto Matriz*vector
//Utiliza el mismo método del Ejercicio 7 para realizar la suma
por columnas

for(i=0; i<N; i++) {
    suma_colum=0;
    #pragma omp parallel firstprivate(suma_colum)
    {
        #pragma omp for
        for(k=0; k<N; k++)
            suma_colum += M1[i][k]*v1[k];

        #pragma omp atomic
        v2[i] += suma_colum;
    }
}

cgt2 = omp_get_wtime();

ncgt= cgt2-cgt1;

//Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
printf("Tiempo(seg.):%11.9f\t / Tamaño Matriz-Vector:
%u\n",ncgt,N);
for(i=0; i<N; i++)
    printf("V2[%d]=%8.6f\n", i, v2[i]);

#else
printf("Tiempo(seg.):%11.9f\t / Tamaño Matriz-Vector:%u\t/
V2[0]=%8.6f / / V2[%d]=%8.6f /\n", ncgt,N, v2[0], N-1, v2[N-1]);
#endif

#ifdef PMV_DYNAMIC
for(i=0; i<N;i++)
    free(M1[i]);
free(M1); // libera el espacio reservado para M1
free(v1); // libera el espacio reservado para v1
free(v2); // libera el espacio reservado para v2
#endif
return 0;
}

```

RESPUESTA:

- En la versión A, el único error notable de compilación que he tenido que solucionar ha sido al olvidar declarar la variable `suma_fila` en el `main`, ya que la había definido con la cláusula `private`, pero no la había declarado inicialmente.

Por lo demás el programa funciona correctamente y ofrece los mismos resultados que en la versión secuencial.

- La versión B, si me ha ocasionado algún problema relacionado con la ejecución del programa. Inicialmente programé esta versión de forma similar al Ejercicio 7, utilizando la directiva `atomic` para evitar que cada thread acceda a la variable `suma_colum` simultáneamente.

El error apareció al declarar la variable `suma_colum` como `private` ya que al ejecutar el programa los resultados eran erróneos. Para arreglarlo, declaré la variable `suma_colum` como `firstprivate` de esta forma, además de declararse como privada, también se inicializa al entrar en la región paralela del bucle `for` con el valor que tuviese inicialmente.

He utilizado las diapositivas de los seminarios 1 y 2 para la corrección de los errores que me han ido surgiendo. También he consultado estos enlaces para aclarar algunas dudas que me han ido surgiendo y tener así una explicación complementaria a la ofrecida en los seminarios.

Enlaces usados: https://lsi.ugr.es/jmantas/pdp/ayuda/omp_ayuda.php

CAPTURAS DE PANTALLA:

NOTA: La ejecución mostrada en las capturas se ha realizado con vectores-matrices dinámicos, su equivalente en globales es igual.

Errores versión A:

```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp p
mv-OpenMP-a.c -o pmv-OpenMP-a -lrt
pmv-OpenMP-a.c: In function 'main':
pmv-OpenMP-a.c:85:38: error: 'suma_fila' undeclared (first use in this
function)
    #pragma omp parallel for private(suma_fila, k)
                                ^
pmv-OpenMP-a.c:85:38: note: each undeclared identifier is reported only
once for each function it appears in
pmv-OpenMP-a.c:86:5: error: for statement expected before '{' token
    {
    ^

```

Errores versión B:

```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp p
mv-OpenMP-b.c -o pmv-OpenMP-b -lrt
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-b 8
Tiempo(seg.):0.000062920 / Tamaño Matriz-Vector:8
V2[0]=9968.880146
V2[1]=9970.560146
V2[2]=9971.880146
V2[3]=9972.840146
V2[4]=9973.440146
V2[5]=9973.680146
V2[6]=9973.560146
V2[7]=9973.080146
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp p
mv-OpenMP-b.c -o pmv-OpenMP-b -lrt
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-b 8
Tiempo(seg.):0.005094684 / Tamaño Matriz-Vector:8
V2[0]=2.040000
V2[1]=1.680000
V2[2]=1.320000
V2[3]=0.960000
V2[4]=0.600000
V2[5]=0.240000
V2[6]=-0.120000
V2[7]=-0.480000

```

Funcionamiento de los programas (errores solucionados):

```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp p
mv-OpenMP-b.c -o pmv-OpenMP-b -lrt
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp p
mv-OpenMP-a.c -o pmv-OpenMP-a -lrt
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-a 8
Tiempo(seg.):0.000011422          / Tamaño Matriz-Vector:8
V2[0]=2.040000
V2[1]=1.680000
V2[2]=1.320000
V2[3]=0.960000
V2[4]=0.600000
V2[5]=0.240000
V2[6]=-0.120000
V2[7]=-0.480000
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-b 8
Tiempo(seg.):0.000011211          / Tamaño Matriz-Vector:8
V2[0]=2.040000
V2[1]=1.680000
V2[2]=1.320000
V2[3]=0.960000
V2[4]=0.600000
V2[5]=0.240000
V2[6]=-0.120000
V2[7]=-0.480000
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-a 11
Tiempo(seg.):0.000002122          / Tamaño Matriz-Vector:11
V2[0]=5.060000
V2[1]=4.400000
V2[2]=3.740000
V2[3]=3.080000
V2[4]=2.420000
V2[5]=1.760000
V2[6]=1.100000
V2[7]=0.440000
V2[8]=-0.220000
V2[9]=-0.880000
V2[10]=-1.540000
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-b 11
Tiempo(seg.):0.000015921          / Tamaño Matriz-Vector:11
V2[0]=5.060000
V2[1]=4.400000
V2[2]=3.740000
V2[3]=3.080000
V2[4]=2.420000
V2[5]=1.760000
V2[6]=1.100000
V2[7]=0.440000
V2[8]=-0.220000
V2[9]=-0.880000
V2[10]=-1.540000

```

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-OpenMP-reduction.c

```

/* pmv-OpenMP-reduction.c
Multiplicación de matriz por vector: v2 = M1*v1
Para compilar usar (-lrt: real time library):
gcc -O2 -fopenmp pmv-OpenMP-reduction.c -o pmv-OpenMP-reduction -lrt

Para ejecutar use: pmv-OpenMP-reduction longitud
*/

```

```

#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()
#include <omp.h>

#define PRINTF_ALL // comentar para quitar el printf ...
                    // que imprime todos los componentes

//Sólo puede estar definida uno de los dos constantes PMV_ (sólo uno
de los ...
//un defines siguientes puede estar descomentado):
//#define PMV_GLOBAL // descomentar para que los vector/matriz sean
variables ...
                    // globales (su longitud no estará
limitada por el ...
                    // tamaño de la pila del programa)
#define PMV_DYNAMIC // descomentar para que los vector/matriz sean
variables ...
                    // dinámicas (memoria reutilizable
durante la ejecución)
#ifdef PMV_GLOBAL
#define MAX 16350 // aprox 2^14
double M1[MAX][MAX], v1[MAX], v2[MAX];
#endif

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

    int i, k;
    double suma_colum;

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

    omp_set_num_threads(2); //para cambiar el n.º threads usadas

    //Leer argumento de entrada (no de componentes del vector/matriz)
    if (argc<2){
        printf("Faltan no componentes del vector/matriz\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)

    #ifdef PMV_GLOBAL
    if (N>MAX) N=MAX;
    #endif
    #ifdef PMV_DYNAMIC
    double **M1, *v1, *v2;
    M1 = (double**) malloc(N*sizeof(double*)); // malloc necesita el
tamaño en bytes
    v1 = (double*) malloc(N*sizeof(double)); //si no hay espacio
suficiente malloc devuelve NULL

```

```

    v2 = (double*) malloc(N*sizeof(double));
    if ( (M1==NULL) || (v1==NULL) || (v2==NULL) ){
        printf("Error en la reserva de espacio para los
vectores/matriz\n");
        exit(-2);
    }
    for(i=0; i<N;i++){
        M1[i] = (double*) malloc(N*sizeof(double));
        if ( M1[i]==NULL ){
            printf("Error en la reserva de espacio para la matriz\n");
            exit(-2);
        }
    }
#endif

//Inicializar matriz y vectores
#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<N; i++){
        v1[i] = N*0.1-i*0.1; //los valores dependen de N
        v2[i]=0;
    }

    #pragma omp for private(k)
    for(i=0; i<N; i++){
        for(k=0; k<N; k++)
            M1[i][k] = N*0.1-i*0.1-k*0.1; //los valores dependen
de N
    }
}

cgt1 = omp_get_wtime();

//Calcular producto Matriz*vector
//Utiliza el mismo método del Ejercicio 7 para realizar la suma
por columnas

for(i=0; i<N; i++) {
    suma_colum=0;
    #pragma omp parallel for reduction(+:suma_colum)

        for(k=0; k<N; k++)
            suma_colum += M1[i][k]*v1[k];

    v2[i] += suma_colum;
}

cgt2 = omp_get_wtime();

ncgt= cgt2-cgt1;

//Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
printf("Tiempo(seg.):%11.9f\t / Tamaño Matriz-Vector:
%u\n",ncgt,N);

```



```

    for(i=0; i<N; i++)
        printf("V2[%d]=%8.6f\n", i, v2[i]);

    #else
        printf("Tiempo(seg.):%11.9f\t / Tamaño Matriz-Vector:%u\t/
V2[0]=%8.6f / / V2[%d]=%8.6f /\n", ncgt,N, v2[0], N-1, v2[N-1]);
    #endif

    #ifdef PMV_DYNAMIC
        for(i=0; i<N;i++)
            free(M1[i]);
        free(M1); // libera el espacio reservado para M1
        free(v1); // libera el espacio reservado para v1
        free(v2); // libera el espacio reservado para v2
    #endif
    return 0;
}

```

RESPUESTA:

El único problema que me ha dado esta versión ha sido en ejecución, ya que me salían resultados erróneos al poner un `#pragma omp for` independiente del bloque paralelo.

Código erróneo:

```

for(i=0; i<N; i++) {
    suma_colum=0;
    #pragma omp parallel reduction(+:suma_colum)
    {
        #pragma omp for
        for(k=0; k<N; k++)
            suma_colum += M1[i][k]*v1[k];

        v2[i] += suma_colum;
    }
}

```

La solución ha sido fácil teniendo en cuenta el recíproco del Ejercicio 7.

CAPTURAS DE PANTALLA:

NOTA: La ejecución mostrada en las capturas se ha realizado con vectores-matrices dinámicos, su equivalente en globales es igual.



```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ gcc -O2 -fopenmp p
mv-OpenMP-reduction.c -o pmv-OpenMP-reduction -lrt
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-reduc
tion 8
Tiempo(seg.):0.000009770          / Tamaño Matriz-Vector:8
V2[0]=1.430000
V2[1]=0.700000
V2[2]=1.210000
V2[3]=0.920000
V2[4]=0.140000
V2[5]=0.340000
V2[6]=-0.350000
V2[7]=-0.320000

```

Funcionamiento de los programas (errores solucionados):

```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-reduc
tion 8
Tiempo(seg.):0.000010152           / Tamaño Matriz-Vector:8
V2[0]=2.040000
V2[1]=1.680000
V2[2]=1.320000
V2[3]=0.960000
V2[4]=0.600000
V2[5]=0.240000
V2[6]=-0.120000
V2[7]=-0.480000
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-OpenMP-reduc
tion 11
Tiempo(seg.):0.000068310           / Tamaño Matriz-Vector:11
V2[0]=5.060000
V2[1]=4.400000
V2[2]=3.740000
V2[3]=3.080000
V2[4]=2.420000
V2[5]=1.760000
V2[6]=1.100000
V2[7]=0.440000
V2[8]=-0.220000
V2[9]=-0.880000
V2[10]=-1.540000

```

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.

Para elegir el código sobre el que realizar el ejercicio he realizado una medición de los tiempos para cada uno de los tres programas en atcgrid utilizando 30000 componentes y 12 threads. Como resultado se puede ver que el que obtiene mejor tiempo de ejecución es el que paraleliza el bucle recorriendo las filas de la matriz.

```

[Eiestudiante20@atcgrid BP2]$ echo 'BP2/pmv-OpenMP-a 30000' | qsub -q
ac
54926.atcgrid
[Eiestudiante20@atcgrid BP2]$ echo 'BP2/pmv-OpenMP-b 30000' | qsub -q
ac
54927.atcgrid
[Eiestudiante20@atcgrid BP2]$ echo 'BP2/pmv-OpenMP-reduction 30000' |
qsub -q ac
54928.atcgrid
[Eiestudiante20@atcgrid BP2]$ cat STDIN.o54926
Tiempo(seg.):0.306646198           / Tamaño Matriz-Vector:30000 / V2[0
]=90004500049.999985 / / V2[29999]=-44995499800.004662 /
[Eiestudiante20@atcgrid BP2]$ cat STDIN.o54927
Tiempo(seg.):0.955205899           / Tamaño Matriz-Vector:30000 / V2[0
]=90004500050.000000 / / V2[29999]=-44995499800.000069 /
[Eiestudiante20@atcgrid BP2]$ cat STDIN.o54928
Tiempo(seg.):1.075289946           / Tamaño Matriz-Vector:30000 / V2[0
]=90004500050.000000 / / V2[29999]=-44995499800.000069 /

```

Es evidente que el caso reduction no iba a ser más eficiente que alguno de los otros dos, puesto que el compilador tarda en montar la estructura de la cláusula reduction y por tanto siempre sería más eficiente su homóloga en columnas sin esa cláusula. En cuanto a

elegir entre filas o columnas, la propia complejidad del segundo caso, utilizando más clausulas que el primero, verifica los resultados obtenidos empíricamente.

Programa pmv-secuencial ejecutado en PC_Local y Atcgrid:

```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-secuencial 1
0000Tiempo(seg.):0.167843858 / Tamaño Matriz-Vector:10000 / V
2[0]=3333833350.000000 / / V2[9999]=-1666166600.000095 /
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_2$ ./pmv-secuencial 3
0000
Tiempo(seg.):1.545052807 / Tamaño Matriz-Vector:30000 / V2[0]
=90004500049.999985 / / V2[29999]=-44995499800.004662 /

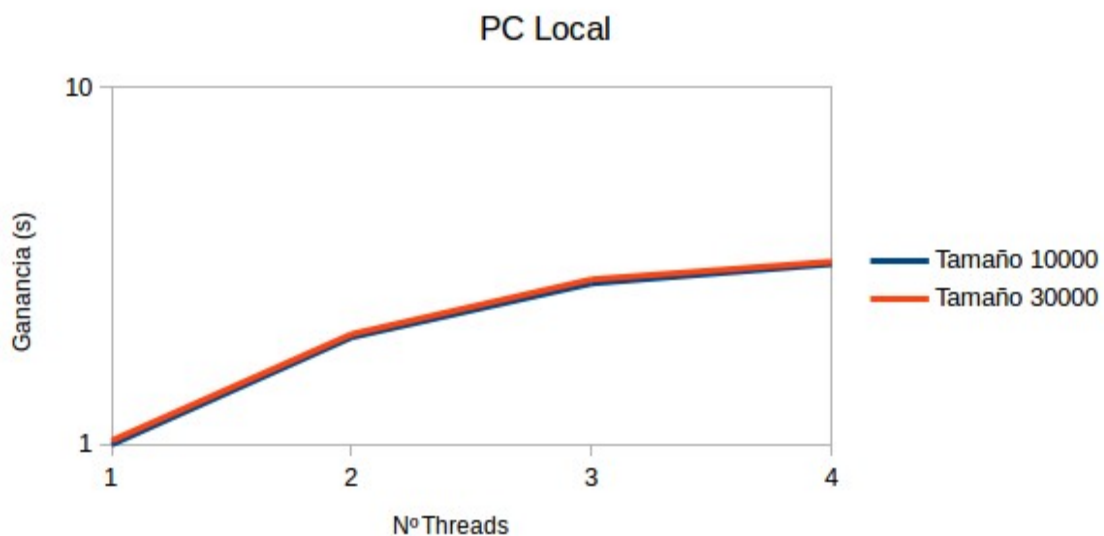
[Eiestudiante20@atcgrid BP2]$ cat STDIN.o55176
Tiempo(seg.):0.143517289 / Tamaño Matriz-Vector:10000 / V2[
0]=3333833350.000000 / / V2[9999]=-1666166600.000095 /
[Eiestudiante20@atcgrid BP2]$ cat STDIN.o55177
Tiempo(seg.):8.544541467 / Tamaño Matriz-Vector:50000 / V2[
0]=416679166750.000000 / / V2[49999]=-208320833000.003296 /
[Eiestudiante20@atcgrid BP2]$

```

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):

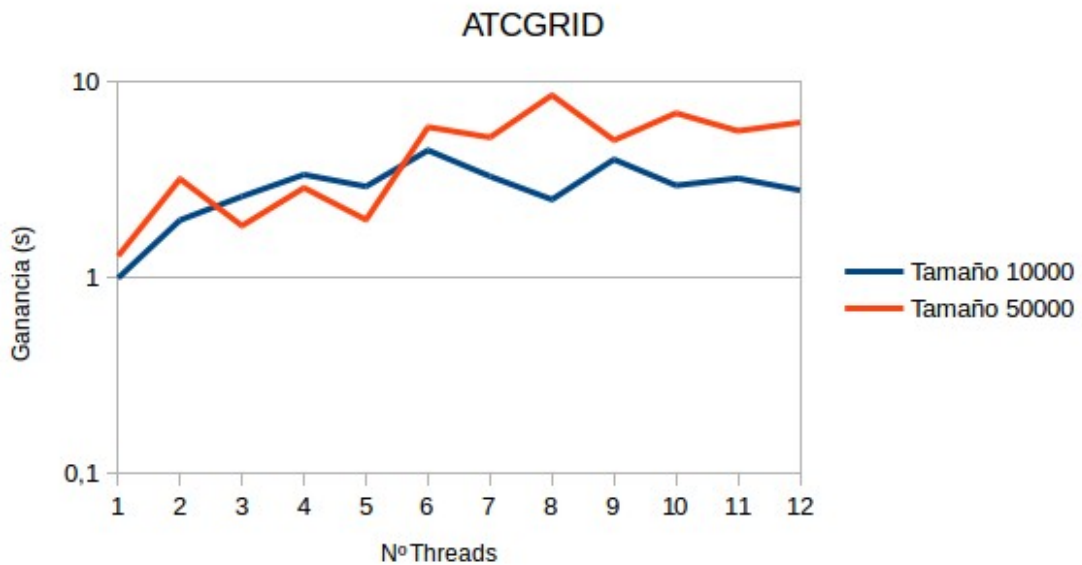
PC Local: Intel® Core™ i5-7200U CPU @ 2.50GHz

Nº de Compon entes vectores /matriz	Threads utilizadas (1-4 threads)	T. secuencial	T. paralelo (versión a) Dinámicos	Ganancia en velocidad
10000	1 thread:	0.167843858	0.167816425	1.0001634703
	2 threads:		0.084072585	1.9964160493
	3 threads:		0.059458884	2.822855841
	4 threads:		0.052448688	3.2001536054
30000	1 thread:	1.545052807	1.505923406	1.0259836595
	2 threads:		0.758193781	2.0378072806
	3 threads:		0.533017872	2.8986885584
	4 threads:		0.475376733	3.2501649739



Atcgrid: Intel® Xeon® CPU E5645 @ 2.40GHz

Nº de Componentes vectores /matriz	Threads utilizadas (1-12 threads)	T. secuencial	T. paralelo (versión a) Dinámicos	Ganancia en velocidad
10000	1 thread:	0.143517289	0.144047145	0.9963216487
	2 threads:		0.072896529	1.9687808318
	3 threads:		0.055209216	2.5995168814
	4 threads:		0.042725779	3.3590327048
	5 thread:		0.049129974	2.9211757572
	6 threads:		0.032119464	4.4682342458
	7 threads:		0.043624837	3.2898068823
	8 threads:		0.057229158	2.5077651675
	9 thread:		0.03576022	4.0133223174
	10 threads:		0.048523761	2.9576703463
	11 threads:		0.044640135	3.2149833104
	12 threads:		0.051361408	2.7942631362
50000	1 thread:	8.544541467	6.60525813	1.293596904
	2 threads:		2.676673401	3.192224148
	3 threads:		4.636166755	1.8430185795
	4 threads:		2.972763106	2.8742759387
	5 thread:		4.321203735	1.9773521433
	6 threads:		1.455489621	5.8705615923
	7 threads:		1.64358237	5.1987302997
	8 threads:		1.001334026	8.5331580123
	9 thread:		1.698935442	5.0293502953
	10 threads:		1.235827643	6.914023582
	11 threads:		1.523876123	5.6071102749
	12 threads:		1.379933197	6.1919964572



COMENTARIOS SOBRE LOS RESULTADOS:

Para empezar cabe destacar que no ha sido posible la realización del ejercicio para valores de cientos de miles puesto que tanto en mi PC como en ATCGRID mataba el proceso. Por ello, he elegido 30000 y 50000 N.º componentes respectivamente para cada caso, valores que pienso que dan resultados buenos para la realización del ejercicio.

Observando las gráficas en general, vemos que a mayor número de threads la ganancia aumenta ya que al repartirse el trabajo en esta versión del programa, a mayor número de componentes, el computador tarda menos.

En ATCGRID con 10000 componentes se obtiene mejor escalabilidad cuanto menor sea el número de threads, con 50000 sin embargo ocurre lo contrario. Esto se debe a que como hemos dicho, la programación en paralelo es mejor cuanto más carga de trabajo tengamos, por ello, con 10000 componentes, las cuales son relativamente pocas, el programa empeora su ganancia de velocidad si usamos más threads.

Los picos que se ven en las gráfica de ATCGRID pueden ser debidos a que la creación y destrucción de threads también afecta al tiempo de ejecución y puede causar que para valores de threads utilizadas cercanos, existan oscilaciones en los resultados obtenidos.