Grai2º curso / 2º cuatr. Grado Ing. Inform.	**Arquitectura de Computadores (AC)** **Cuaderno de prácticas.** **Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP** Estudiante (nombre y apellidos): Salvador Romero Cortés Grupo de prácticas y profesor de prácticas: A1 Niceto Rafael Luque Fecha de entrega: Fecha evaluación en clase:

Antes de comenzar a realizar el trabajo de este cuaderno consultar el fichero con los normas de prácticas que se encuentra en SWAD

Ejercicios basados en los ejemplos del seminario práctico

1. **(a)** Añadir la cláusula `default(none)` a la directiva `parallel` del ejemplo del seminario `shared-clause.c`? ¿Qué ocurre? ¿A qué se debe? **(b)** Resolver el problema generado sin eliminar `default(none)`. Incorporar el código con la modificación al cuaderno de prácticas. (Añadir capturas de pantalla que muestren lo que ocurre)

   RESPUESTA:
   Produce un error de compilación. Nos informa de que no hemos especificado la variable n en la sección paralela.

   CAPTURA CÓDIGO FUENTE: `shared-clauseModificado.c`

```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), 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:

```
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/
bp2/ejer1] 2021-04-20 Tuesday
$g++ -o shared-clauseModificado shared-clauseModificado.c -fopenmp -O2
shared-clauseModificado.c: In function ‘int main()’:
shared-clauseModificado.c:16:5: error: ‘n’ not specified in enclosing ‘para
llel’
   16 |     for (i=0; i < n; i++) a[i] += i;
      |     ^~~
shared-clauseModificado.c:15:13: error: enclosing ‘parallel’
   15 |     #pragma omp parallel for shared(a), default(none)
      |             ^~~
```

Para solucionarlo podemos incluir n en la directiva shared.

```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]);
}
```

Si ahora ejecutamos vemos que funciona correctamente.

```
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/
bp2/ejer1] 2021-04-20 Tuesday
$g++ -o shared-clauseModificado shared-clauseModificado.c -fopenmp -O2
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/
bp2/ejer1] 2021-04-20 Tuesday
$./shared-clauseModificado
Después de parallel for:
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
a[6] = 13
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/
bp2/ejer1] 2021-04-20 Tuesday
$
```

2. **(a)** Añadir a lo necesario a `private-clause.c` para que imprima suma fuera de la región `parallel`. Inicializar `suma` dentro del `parallel` a un valor distinto de 0. Ejecutar varias veces el código ¿Qué imprime el código fuera del `parallel`? (mostrar lo que ocurre con una captura de pantalla) Razonar respuesta. **(b)** Modificar el código del apartado (a) para que se inicialice `suma` fuera del `parallel` en lugar de dentro ¿Qué ocurre? Comparar todo lo que imprime el código ahora con la salida en (a) (mostrar la salida con una captura de pantalla) Razonar respuesta.

(a) RESPUESTA:

La suma fuera del parallel muestra 0 siempre porque los valores se modificaron en una copia privada de cada thread. Esto se debe a la clausula private(suma). Por tanto, en el programa principal no se modifica la suma.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado_a.c`

```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;

    #pragma omp parallel private(suma)
    {
        suma = 1310;
        #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");

    printf("--- fuera region parallel --- \n thread %d suma=%d\n", omp_get_thread_num(), suma);
}
```

CAPTURAS DE PANTALLA:

```
[SalvadorRomeroCortés salva@pop-os:~/Uni/2Info-C2/AC/bp2/ejer2] 2021-04-22 jueves
$g++ -o private-clauseModificado_a private-clauseModificado_a.c -O2 -fopenmp
[SalvadorRomeroCortés salva@pop-os:~/Uni/2Info-C2/AC/bp2/ejer2] 2021-04-22 jueves
$./private-clauseModificado_a
thread 5 suma a[5]/thread 0 suma a[0]/thread 3 suma a[3]/thread 2 suma a[2]/thread 6 suma a[6]/thread 1 suma a[1]/thread 4 suma a[4]/
* thread 5 suma=1315
* thread 0 suma=1310
* thread 2 suma=1312
* thread 3 suma=1313
* thread 1 suma=1311
* thread 4 suma=1314
* thread 6 suma=1316
* thread 9 suma=1310
* thread 8 suma=1310
* thread 7 suma=1310
* thread 11 suma=1310
* thread 10 suma=1310
--- fuera region parallel ---
 thread 0 suma=0
[SalvadorRomeroCortés salva@pop-os:~/Uni/2Info-C2/AC/bp2/ejer2] 2021-04-22 jueves
$./private-clauseModificado_a
thread 0 suma a[0]/thread 4 suma a[4]/thread 3 suma a[3]/thread 6 suma a[6]/thread 2 suma a[2]/thread 1 suma a[1]/thread 5 suma a[5]/
* thread 2 suma=1312
* thread 5 suma=1315
* thread 1 suma=1311
* thread 10 suma=1310
* thread 6 suma=1316
* thread 9 suma=1310
* thread 3 suma=1313
* thread 8 suma=1310
* thread 0 suma=1310
* thread 7 suma=1310
* thread 11 suma=1310
* thread 4 suma=1314
--- fuera region parallel ---
 thread 0 suma=0
```

(b) RESPUESTA:
Muestra siempre la suma a la que lo inicializamos.
Esto ocurre porque la variable suma tiene una copia privada para cada hebra y por tanto cuando se imprime fuera de esta sale el resultado de la inicialización.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado_b.c`

```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 = 1310;

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

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

CAPTURAS DE PANTALLA:

```
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer2] 2021-04-24 Saturday
$g++ -o private-clauseModificado_b private-clauseModificado_b.c -O2 -fopenmp
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer2] 2021-04-24 Saturday
$./private-clauseModificado_b
thread 1 suma a[1]/thread 4 suma a[4]/thread 3 suma a[3]/thread 0 suma a[0]/thread 6 suma a[6]/thread 2 suma a[2]/thread 5 suma
 a[5]/
* thread 4 suma=1545810644
* thread 6 suma=1545810646
* thread 2 suma=1545810642
* thread 1 suma=1545810641
* thread 5 suma=1545810645
* thread 3 suma=1545810643
* thread 7 suma=1545810640
* thread 0 suma=8
--- fuera region parallel ---
 thread 0 suma=1310
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer2] 2021-04-24 Saturday
$./private-clauseModificado_b
thread 3 suma a[3]/thread 1 suma a[1]/thread 5 suma a[5]/thread 2 suma a[2]/thread 4 suma a[4]/thread 6 suma a[6]/thread 0 suma
 a[0]/
* thread 2 suma=-1334123822
* thread 4 suma=-1334123820
* thread 7 suma=-1334123824
* thread 3 suma=-1334123821
* thread 5 suma=-1334123819
* thread 6 suma=-1334123818
* thread 0 suma=8
* thread 1 suma=-1334123823
--- fuera region parallel ---
 thread 0 suma=1310
```

Vemos que en comparación con el apartado a, en el b siempre muestra el valor al que lo inicializamos, a pesar de que cada thread lo modifica. En el apartado a, siempre muestra 0 en lugar del resultado.

Esto se debe a que la inicialización de A se realiza en una copia privada que posee cada thread que lo ejecuta, mientras que en el apartado B, la inicialización se realiza fuera y luego en la copia privada se modifica, sin afectar al valor original al que se inicializó. Notar también que como hemos inicializado la variable suma fuera del parallel en el apartado B, se trabaja con valores basura.

3. **(a)** Eliminar la cláusula `private(suma)` en `private-clause.c`. Ejecutar el código resultante. ¿Qué ocurre? **(b)** ¿A qué es debido?

RESPUESTA:

Vemos que el resultado no es correcto, puesto que las hebras modifican la misma dirección de memoria, no se copia. Por lo tanto, si varias hebras lo hacen al mismo tiempo se pierden datos y no se produce el resultado correcto.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado3.c`

```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;

    #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");
}
```

CAPTURAS DE PANTALLA:

```
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer3] 2021-04-24 Saturday
$g++ -o private-clauseModificado3 private-clauseModificado3.c -O2 -fopenmp
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer3] 2021-04-24 Saturday
$./private-clauseModificado3
thread 0 suma a[0] / thread 3 suma a[3] / thread 4 suma a[4] / thread 6 suma a[6] / thread 2 suma a[2] / thread 1 suma a[1] / thread 5 suma a[5] /
* thread 3 suma= 0
* thread 0 suma= 0
* thread 1 suma= 0
* thread 4 suma= 0
* thread 5 suma= 0
* thread 6 suma= 0
* thread 7 suma= 0
* thread 2 suma= 0
```

4. En la ejecución de `firstlastprivate.c` de la pag. 21 del seminario se imprime un 6 fuera de la región `parallel`. **(a)** Cambiar el tamaño del vector a 10. Razonar lo que imprime el código en su PC con esta modificación. (añadir capturas de pantalla que muestren lo que ocurre). **(b)** Sin cambiar el tamaño del vector ¿podría imprimir el código otro valor? Razonar respuesta (añadir capturas de pantalla que muestren lo que ocurre).

(a) RESPUESTA:

Imprime 9 porque se copia a la variable secuencial el valor que deja el último thread de la ejecución del for. Esto ocurre por la cláusula `lastprivate` que copia el valor de la última iteración del for a la variable fuera del entorno paralelo.

CAPTURAS DE PANTALLA:

```
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer4] 2021-04-24 Saturday
$g++ -o firstlastprivate firstlastprivate.c -O2 -fopenmp
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer4] 2021-04-24 Saturday
$./firstlastprivate
 thread 6 suma a[8] suma = 8
 thread 4 suma a[6] suma = 6
 thread 5 suma a[7] suma = 7
 thread 3 suma a[5] suma = 5
 thread 1 suma a[2] suma = 2
 thread 1 suma a[3] suma = 5
 thread 7 suma a[9] suma = 9
 thread 0 suma a[0] suma = 0
 thread 0 suma a[1] suma = 1
 thread 2 suma a[4] suma = 4

Fuera de la construcción parallel suma=9
```

(b) RESPUESTA:

No puede imprimir otro valor, siempre va a imprimir `n-1` puesto que la última iteración del bucle suma `a[n-1]` que es igual a `n-1`.

CAPTURAS DE PANTALLA:

```
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer4] 2021-04-24 Saturday
$./firstlastprivate
 thread 0 suma a[0] suma = 0
 thread 0 suma a[1] suma = 1
 thread 3 suma a[5] suma = 5
 thread 7 suma a[9] suma = 9
 thread 2 suma a[4] suma = 4
 thread 6 suma a[8] suma = 8
 thread 5 suma a[7] suma = 7
 thread 4 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=9
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer4] 2021-04-24 Saturday
$./firstlastprivate
 thread 6 suma a[8] suma = 8
 thread 5 suma a[7] suma = 7
 thread 4 suma a[6] suma = 6
 thread 7 suma a[9] suma = 9
 thread 2 suma a[4] suma = 4
 thread 3 suma a[5] suma = 5
 thread 1 suma a[2] suma = 2
 thread 1 suma a[3] suma = 5
 thread 0 suma a[0] suma = 0
 thread 0 suma a[1] suma = 1

Fuera de la construcción parallel suma=9
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer4] 2021-04-24 Saturday
$./firstlastprivate
 thread 2 suma a[4] suma = 4
 thread 1 suma a[2] suma = 2
 thread 1 suma a[3] suma = 5
 thread 3 suma a[5] suma = 5
 thread 7 suma a[9] suma = 9
 thread 4 suma a[6] suma = 6
 thread 6 suma a[8] suma = 8
 thread 0 suma a[0] suma = 0
 thread 0 suma a[1] suma = 1
 thread 5 suma a[7] suma = 7

Fuera de la construcción parallel suma=9
```

Vemos que sin cambiar el tamaño del vector, siempre imprime 9.

5. **(a)** ¿Qué se observa en los resultados de ejecución de `copyprivate-clause.c` cuando se elimina la cláusula `copyprivate(a)` en la directiva `single`? **(b)** ¿A qué cree que es debido? (añadir una captura de pantalla que muestre lo que ocurre)

RESPUESTA:

Ocurre que no se comparte el valor introducido por el usuario al resto de threads, por lo tanto sólo un elemento del vector tiene el valor correcto.

CAPTURA CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

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

int main(){
    int n=9, i, b[n];

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

```
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer5] 2021-04-24 Saturday
$g++ -o copyprivate-clauseModificado copyprivate-clauseModificado.c -O2 -fopenmp
[SalvadorRomeroCortés salva@DESKTOP-7KBAEK1:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer5] 2021-04-24 Saturday
$./copyprivate-clauseModificado

Introduce valor de inicialización a: 1310

Single ejecutada por el thread 5
Después de la región parallel:
b[0] = 21994    b[1] = 21994    b[2] = 0        b[3] = 0        b[4] = 0        b[5] = 0
[6] = 1310      b[7] = 0        b[8] = 0
```

6. En el ejemplo `reduction-clause.c` sustituya `suma=0` por `suma=10`. ¿Qué resultado se imprime ahora? Justifique el resultado (añada capturas de pantalla que muestren lo que ocurre)

RESPUESTA:

Ahora el resultado es la suma(45) más el valor al que hemos inicializado (10). Es decir, 55. Esto ocurre porque la suma se distribuye entre los distintos threads y al final con reduction se suma a la variable, que ya tenía un valor previo: 10.

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado.c`

```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){
        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:

```
[SalvadorRomeroCortés salva@DESKTOP-1UETQFU:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer6] 2021-04-24 Saturday
$g++ -o reduction-clauseModificado reduction-clauseModificado.c -O2 -fopenmp
[SalvadorRomeroCortés salva@DESKTOP-1UETQFU:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer6] 2021-04-24 Saturday
$./reduction-clauseModificado 10
Tras 'parallel' suma = 55
```

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 añadir más directivas de trabajo compartido (añada capturas de pantalla que muestren lo que ocurre).

RESPUESTA:

Sumo en una variable local de cada thread el resultado y al final se suma a la variable suma en una región de exclusión mutua.

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado7.c`

```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;

    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
    {
        int suma_local = 0;

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

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

CAPTURAS DE PANTALLA:

```
[SalvadorRomeroCortés salva@DESKTOP-1UETQFU:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer7] 2021-04-24 Saturday
$g++ -o reduction-clauseModificado7 reduction-clauseModificado7.c -O2 -fopenmp
[SalvadorRomeroCortés salva@DESKTOP-1UETQFU:/mnt/c/Users/pingu/2Info-C2/AC/bp2/ejer7] 2021-04-24 Saturday
$./reduction-clauseModificado7 10
Tras 'parallel' suma = 45
```

Resto de ejercicios (usar en atcgrid la cola ac a no ser que se tenga que usar atcgrid4)

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):

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

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).

CAPTURA CÓDIGO FUENTE: `pmv-secuencial.c`

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

//#define VECTOR_GLOBAL

#define VECTOR_DYNAMIC

#ifdef VECTOR_GLOBAL
#define MAX 33554432

double matriz[MAX][MAX], v1[MAX], v3[MAX];
#endif

int main(int argc, char** argv) {
    int i,j;

    double tiempo;
    struct timespec cgt1, cgt2;

    if (argc != 2) {
        printf("Argumentos incorrectos. Uso: binario <tamaño>");
        exit(-1);
    }

    int dimension = atoi(argv[1]);

//reserva de memoria dinamica
#ifdef VECTOR_DYNAMIC
    double** m = (double**)malloc(dimension * sizeof(double *));
    double* v1 = (double*)malloc(dimension * sizeof(double));
    double* v3 = (double*)malloc(dimension * sizeof(double));

    for (i = 0; i < dimension; i++) {
        m[i] = (double*)malloc(dimension * sizeof(double));
    }
#endif

    //inicializacion de datos de la matriz y los vectores
    for (i = 0; i < dimension; i++) {
        for (j = 0; j < dimension; j++) {
            m[i][j] = 0.1*i - 0.1*j;
        }
        v1[i] = 0.1*i;
        v3[i] = 0;
    }

    // calculo
    clock_gettime(CLOCK_REALTIME, &cgt1);
    for (i = 0; i < dimension; i++) {
        for (j = 0; j < dimension; j++) {
            v3[i] += m[i][j] * v1[j];
        }
    }
```

```
    clock_gettime(CLOCK_REALTIME, &cgt2);
    tiempo = (double)(cgt2.tv_sec - cgt1.tv_sec) +
        (double)((cgt2.tv_nsec - cgt1.tv_nsec) / (1.e+9));
    // muestra de resultados
    if (dimension <= 10) {
        printf("Matriz: \n");
        for (i = 0; i < dimension; i++) {
            for (j = 0; j < dimension; j++) {
                printf("%f\t", m[i][j]);
            }
            printf("\n");
        }
        printf("\n");
        printf("Vector producto: \n");
        for (i = 0; i < dimension; i++)
            printf("%f", v1[i]);
        printf("\n");

        printf("Vector resultado: \n");
        for (i = 0; i < dimension; i++) {
            printf("%f", v3[i]);
        }
        printf("\n");

    }
    else {
        printf("Matriz m[0][0] = %f <--> m[%d][%d] = %f\n",
            m[0][0], dimension - 1, dimension - 1, m[dimension - 1][dimension - 1]);
        printf("V1[0] = %f <--> V1[%d] = %f\n",
            v1[0], dimension - 1, v1[dimension - 1]);
        printf("Resultado V3[0] = %f <--> V3[%d] = %f\n",
            v3[0], dimension - 1, v3[dimension - 1]);
    }
    printf("Tiempo: %f11.9\n", tiempo);
//liberacion de memoria dinamica
#ifdef VECTOR_DYNAMIC
    for (i = 0; i < dimension; i++)
        free(m[i]);
    free(m);
    free(v1);
    free(v3);
#endif
}
```

CAPTURAS DE PANTALLA:

```
[a1estudiante23@atcgrid ejer8]$ g++ -O2 -o pmv-secuencial pmv-secuencial.c
[a1estudiante23@atcgrid ejer8]$ srun -p ac -A ac ./pmv-secuencial 8
Matriz:
0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000       -0.600000       -0.700000
0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000       -0.600000
0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000
0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000
0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000
0.500000        0.400000        0.300000        0.200000        0.100000        0.000000 -0.100000      -0.200000
0.600000        0.500000        0.400000        0.300000        0.200000        0.100000 0.000000       -0.100000
0.700000        0.600000        0.500000        0.400000        0.300000        0.200000 0.100000       0.000000

Vector producto:
0.0000000.1000000.2000000.3000000.4000000.5000000.6000000.700000
Vector resultado:
-1.400000-1.120000-0.840000-0.560000-0.280000-0.0000000.2800000.560000
Tiempo: 0.00000011.9
[a1estudiante23@atcgrid ejer8]$ srun -p ac -A ac ./pmv-secuencial 11
Matriz m[0][0] = 0.000000 <--> m[10][10] = 0.000000
V1[0] = 0.000000 <--> V1[10] = 1.000000
Resultado V3[0] = -3.850000 <--> V3[10] = 1.650000
Tiempo: 0.00000011.9
```

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):

a. una primera que paralelice el bucle que recorre las filas de la matriz y

b. una segunda que paralelice el bucle que recorre las columnas.

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

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

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).

CAPTURA CÓDIGO FUENTE : `pmv-OpenMP-a.c`

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

//#define VECTOR_GLOBAL

#define VECTOR_DYNAMIC

#ifdef VECTOR_GLOBAL
#define MAX 33554432

double matriz[MAX][MAX], v1[MAX], v3[MAX];
#endif

int main(int argc, char** argv) {
    int i,j;

    double tiempo;
    double inicio, final;

    if (argc != 2) {
        printf("Argumentos incorrectos. Uso: binario <tamaño>");
        exit(-1);
    }

    int dimension = atoi(argv[1]);

//reserva de memoria dinamica
#ifdef VECTOR_DYNAMIC
    double** m = (double**)malloc(dimension * sizeof(double *));
    double* v1 = (double*)malloc(dimension * sizeof(double));
    double* v3 = (double*)malloc(dimension * sizeof(double));

    for (i = 0; i < dimension; i++) {
        m[i] = (double*)malloc(dimension * sizeof(double));
    }
#endif
```

```
#pragma omp parallel
{
    //inicializacion de datos de la matriz y los vectores
    #pragma omp for
    for (i = 0; i < dimension; i++) {
        for (j = 0; j < dimension; j++) {
            m[i][j] = 0.1*i - 0.1*j;
        }
        v1[i] = 0.1*i;
        v3[i] = 0;
    }

    // calculo
    #pragma omp single
    inicio = omp_get_wtime();

    #pragma omp for private(i,j)
    for (i = 0; i < dimension; i++) {
        for (j = 0; j < dimension; j++) {
            v3[i] += m[i][j] * v1[j];
        }
    }
    #pragma omp single
    {
        final = omp_get_wtime();
        tiempo = final - inicio;
    }
}
// muestra de resultados
if (dimension <= 10) {
    printf("Matriz: \n");
    for (i = 0; i < dimension; i++) {
        for (j = 0; j < dimension; j++) {
            printf("%f\t", m[i][j]);
        }
        printf("\n");
    }
    printf("\n");
    printf("Vector producto: \n");
    for (i = 0; i < dimension; i++)
        printf("%f", v1[i]);
    printf("\n");

    printf("Vector resultado: \n");
    for (i = 0; i < dimension; i++) {
        printf("%f", v3[i]);
    }
    printf("\n");

}
```

```
    else {
        printf("Matriz m[0][0] = %f <--> m[%d][%d] = %f\n",
            m[0][0], dimension - 1, dimension - 1, m[dimension - 1][dimension - 1]);
        printf("V1[0] = %f <--> V1[%d] = %f\n",
            v1[0], dimension - 1, v1[dimension - 1]);
        printf("Resultado V3[0] = %f <--> V3[%d] = %f\n",
            v3[0], dimension - 1, v3[dimension - 1]);
    }
    printf("Tiempo: %f11.9\n", tiempo);
//liberacion de memoria dinamica
#ifdef VECTOR_DYNAMIC
    for (i = 0; i < dimension; i++)
        free(m[i]);
    free(m);
    free(v1);
    free(v3);
#endif
}
```

CAPTURA CÓDIGO FUENTE: `pmv-OpenMP-b.c`

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

//#define VECTOR_GLOBAL

#define VECTOR_DYNAMIC

#ifdef VECTOR_GLOBAL
#define MAX 33554432

double matriz[MAX][MAX], v1[MAX], v3[MAX];
#endif

int main(int argc, char** argv) {
    int i,j;

    double tiempo;
    double inicio, final;

    if (argc != 2) {
        printf("Argumentos incorrectos. Uso: binario <tamaño>");
        exit(-1);
    }

    int dimension = atoi(argv[1]);

//reserva de memoria dinamica
#ifdef VECTOR_DYNAMIC
    double** m = (double**)malloc(dimension * sizeof(double *));
    double* v1 = (double*)malloc(dimension * sizeof(double));
    double* v3 = (double*)malloc(dimension * sizeof(double));

    for (i = 0; i < dimension; i++) {
        m[i] = (double*)malloc(dimension * sizeof(double));
    }
#endif

    //inicializacion de datos de la matriz y los vectores
    for (i = 0; i < dimension; i++) {
        v1[i] = 0.1*i;
        v3[i] = 0;
```

```
    #pragma omp parallel for
        for (j = 0; j < dimension; j++) {
            m[i][j] = 0.1*i - 0.1*j;
        }
    }

    // calculo
    inicio = omp_get_wtime();
    int aux = 0;
    for (i = 0; i < dimension; i++) {
        #pragma omp parallel firstprivate(aux)
        {
            #pragma omp for
            for (j = 0; j < dimension; j++) {
                aux += m[i][j] * v1[j];
            }

            #pragma omp atomic
            v3[i] += aux;
        }
    }

    final = omp_get_wtime();
    tiempo = final - inicio;

    // muestra de resultados
    if (dimension <= 10) {
        printf("Matriz: \n");
        for (i = 0; i < dimension; i++) {
            for (j = 0; j < dimension; j++) {
                printf("%f\t", m[i][j]);
            }
            printf("\n");
        }
        printf("\n");
        printf("Vector producto: \n");
        for (i = 0; i < dimension; i++)
            printf("%f", v1[i]);
        printf("\n");
```

```
        printf("Vector resultado: \n");
        for (i = 0; i < dimension; i++) {
            printf("%f", v3[i]);
        }
        printf("\n");

    }
    else {
        printf("Matriz m[0][0] = %f <--> m[%d][%d] = %f\n",
            m[0][0], dimension - 1, dimension - 1, m[dimension - 1][dimension - 1]);
        printf("V1[0] = %f <--> V1[%d] = %f\n",
            v1[0], dimension - 1, v1[dimension - 1]);
        printf("Resultado V3[0] = %f <--> V3[%d] = %f\n",
            v3[0], dimension - 1, v3[dimension - 1]);
    }
    printf("Tiempo: %f11.9\n", tiempo);
//liberacion de memoria dinamica
#ifdef VECTOR_DYNAMIC
    for (i = 0; i < dimension; i++)
        free(m[i]);
    free(m);
    free(v1);
    free(v3);
#endif
}
```

RESPUESTA:

Versión A:

No he tenido ningún error de compilación ni de ejecución.

Versión B:

No he tenido ningún error de compilación ni de ejecución.

CAPTURAS DE PANTALLA:

Versión A:

```
[a1estudiante23@atcgrid ejer9]$ gcc -O2 -fopenmp -o pmv-openMP-a pmv-openMP-a.c
[a1estudiante23@atcgrid ejer9]$ srun -pac -Aac -n1 -c12 --hint=nomultithread ./pmv-openMP-a 8
Matriz:
0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000       -0.600000       -0.700000
0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000       -0.600000
0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000
0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000
0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000
0.500000        0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000
0.600000        0.500000        0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000
0.700000        0.600000        0.500000        0.400000        0.300000        0.200000        0.100000        0.000000

Vector producto:
0.0000000.1000000.2000000.3000000.4000000.5000000.6000000.700000
Vector resultado:
-1.400000-1.120000-0.840000-0.560000-0.280000-0.0000000.2800000.560000
Tiempo: 0.00327311.9
[a1estudiante23@atcgrid ejer9]$ srun -pac -Aac -n1 -c12 --hint=nomultithread ./pmv-openMP-a 11
Matriz m[0][0] = 0.000000 <--> m[10][10] = 0.000000
V1[0] = 0.000000 <--> V1[10] = 1.000000
Resultado V3[0] = -3.850000 <--> V3[10] = 1.650000
Tiempo: 0.00321011.9
```

Versión B:

```
[a1estudiante23@atcgrid ejer9]$ gcc -O2 -fopenmp -o pmv-openMP-b pmv-openMP-b.c
[a1estudiante23@atcgrid ejer9]$ srun -pac -Aac -n1 -c12 --hint=nomultithread ./pmv-openMP-b 8
Matriz:
0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000       -0.600000       -0.700000
0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000       -0.600000
0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000
0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000
0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000
0.500000        0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000
0.600000        0.500000        0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000
0.700000        0.600000        0.500000        0.400000        0.300000        0.200000        0.100000        0.000000

Vector producto:
0.0000000.1000000.2000000.3000000.4000000.5000000.6000000.700000
Vector resultado:
0.0000000.0000000.0000000.0000000.0000000.0000000.0000000.000000
Tiempo: 0.00006411.9
[a1estudiante23@atcgrid ejer9]$ srun -pac -Aac -n1 -c12 --hint=nomultithread ./pmv-openMP-b 11
Matriz m[0][0] = 0.000000 <--> m[10][10] = 0.000000
V1[0] = 0.000000 <--> V1[10] = 1.000000
Resultado V3[0] = -1.000000 <--> V3[10] = 0.000000
Tiempo: 0.00008211.9
```

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).

CAPTURA CÓDIGO FUENTE: `pmv-OpenmMP-reduction.c`

La parte que difiere del código del apartado b del ejercicio es la siguiente:

```
// calculo
inicio = omp_get_wtime();
int aux = 0;
for (i = 0; i < dimension; i++) {
    #pragma omp parallel for reduction(+:aux)
        for (j = 0; j < dimension; j++) {
            aux += m[i][j] * v1[j];
        }
        v3[i] += aux;
}
```

RESPUESTA:
No he tenido ningún error de ningún tipo.

CAPTURAS DE PANTALLA:

```
[a1estudiante23@atcgrid ejer10]$ gcc -O2 -fopenmp -o pmv-openMP-reduction pmv-openMP-reduction.c
[a1estudiante23@atcgrid ejer10]$ srun -pac -Aac -n1 -c12 --hint=nomultithread ./pmv-openMP-reduction 8
Matriz:
0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000       -0.600000       -0.700000
0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000       -0.600000
0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000       -0.500000
0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000       -0.400000
0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000       -0.300000
0.500000        0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000       -0.200000
0.600000        0.500000        0.400000        0.300000        0.200000        0.100000        0.000000        -0.100000
0.700000        0.600000        0.500000        0.400000        0.300000        0.200000        0.100000        0.000000

Vector producto:
0.0000000.1000000.2000000.3000000.4000000.5000000.6000000.700000
Vector resultado:
0.0000000.0000000.0000000.0000000.0000000.0000000.0000000.000000
Tiempo: 0.02699811.9
[a1estudiante23@atcgrid ejer10]$ srun -pac -Aac -n1 -c12 --hint=nomultithread ./pmv-openMP-reduction 11
Matriz m[0][0] = 0.000000 <--> m[10][10] = 0.000000
V1[0] = 0.000000 <--> V1[10] = 1.000000
Resultado V3[0] = -1.000000 <--> V3[10] = -1.000000
Tiempo: 0.03986411.9
```

11. Realizar una tabla y una gráfica que permitan comparar la escalabilidad (ganancia en velocidad en función del número de cores) en atcgrid4, en uno de los nodos de la cola ac y en su PC 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.

CAPTURAS DE PANTALLA (que justifique el código elegido):

```
[SalvadorRomeroCortés salva@pop-os:~/Uni/2Info-C2/AC/bp2/ejer11/pruebas_velocidad] 2021-04-26 lunes
$./pmv-openMP-a 100
Matriz m[0][0] = 0.000000 <--> m[99][99] = 0.000000
V1[0] = 0.000000 <--> V1[99] = 9.900000
Resultado V3[0] = -3283.500000 <--> V3[99] = 1617.000000
Tiempo: 0.00000511.9
[SalvadorRomeroCortés salva@pop-os:~/Uni/2Info-C2/AC/bp2/ejer11/pruebas_velocidad] 2021-04-26 lunes
$./pmv-openMP-b 100
Matriz m[0][0] = 0.000000 <--> m[99][99] = 0.000000
V1[0] = 0.000000 <--> V1[99] = 9.900000
Resultado V3[0] = -3243.000000 <--> V3[99] = 1564.000000
Tiempo: 0.00048411.9
[SalvadorRomeroCortés salva@pop-os:~/Uni/2Info-C2/AC/bp2/ejer11/pruebas_velocidad] 2021-04-26 lunes
$./pmv-openMP-reduction 100
Matriz m[0][0] = 0.000000 <--> m[99][99] = 0.000000
V1[0] = 0.000000 <--> V1[99] = 9.900000
Resultado V3[0] = -25944.000000 <--> V3[99] = -833430.000000
Tiempo: 0.00050611.9
```

Vemos que el que tarda menos tiempo es el código A.

JUSTIFICAR AHORA EN BASE AL CÓDIGO LA DIFERENCIA EN TIEMPOS:

Podemos descartar la b porque es una hebra que lanza al resto y por tanto tiene menos paralelismo con respecto al resto y por eso es más lenta. En segundo lugar, reduction porque se organiza mejor, pero tiene que "recoger" el resultado. En primer lugar, el código A porque se reparte de manera que se obtiene el máximo paralelismo.

CAPTURA DE PANTALLA del script `pmv-OpenmMP-script.sh`

```
#!/bin/bash
#Órdenes para el Gestor de carga de un trabajo:
#1. Asigna al trabajo un nombre
#SBATCH --job-name=SumaLocal
#2. Asignar el trabajo a una partición (cola)
#SBATCH --partition=ac
#3. Asignar el trabajo a un account
#SBATCH --account=ac

#Obtener información de las variables del entorno del Gestor de carga de trabajo
echo "Id. usuario del trabajo: $SLURM_JOB_USER"
echo "Id. del trabajo: $SLURM_JOBID"
echo "Nombre del trabajo especificado por usuario: $SLURM_JOB_NAME"
echo "Directorio de trbajo (en el que se ejecuta el script): $SLURM_SUBMIT_DIR"
echo "Cola: $SLURM_JOB_PARTITION"
echo "Nodo que ejecuta este trabajo: $SLURM_SUBMIT_HOST"
echo "Nº de nodos asignados al trabajo: $SLURM_JOB_NUM_NODES"
echo "Nodos asignados al trabajo: $SLURM_JOB_NODELIST"
echo "CPUs por nodo: $SLURM_JOB_CPUS_PER_NODE"
#Instrucciones del script para ejecutar código:

echo "SECUENCIAL"
echo "Tamaño: 7500"
srun ./secuencial 7500

echo "Tamaño: 50000"
srun ./secuencial 50000

echo "PARALELO"
echo "Tamaño: 7500"
for i in `seq 1 1 32`; do
echo " -- Numero de cores: $i -- "
EXPORT OMP_NUM_THREADS=$i
srun ./paralelo 7500
done

echo "Tamaño: 50000"
for i in `seq 1 1 32`; do
export OMP_NUM_THREADS=$i
echo " -- Numero de cores: $i -- "
srun ./paralelo 50000
done
```

CAPTURAS DE PANTALLA (mostrar la ejecución en atcgrid – envío(s) a la cola):

Atcgrid[1-3]:

```
[a1estudiante23@atcgrid ejer11]$ sbatch -pac -Aac -n1 -c12 --hint=nomultithread pmv-OpenMP-script.sh
Submitted batch job 98448
[a1estudiante23@atcgrid ejer11]$ cat slurm-98448.out
Id. usuario del trabajo: a1estudiante23
Id. del trabajo: 98448
Nombre del trabajo especificado por usuario: SumaLocal
Directorio de trbajo (en el que se ejecuta el script): /home/a1estudiante23/bp2/ejer11
Cola: ac
Nodo que ejecuta este trabajo: atcgrid.ugr.es
N° de nodos asignados al trabajo: 1
Nodos asignados al trabajo: atcgrid1
CPUs por nodo: 24
SECUENCIAL
Tamaño: 7500
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.08042311.9
Tamaño: 50000
Matriz m[0][0] = 0.000000 <--> m[49999][49999] = 0.000000
V1[0] = 0.000000 <--> V1[49999] = 4999.900000
Resultado V3[0] = -416654166750.000000 <--> V3[49999] = 208320833500.003296
Tiempo: 5.54789311.9
PARALELO
Tamaño: 7500
 -- Numero de cores: 1 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.08227411.9
 -- Numero de cores: 2 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.04625111.9
 -- Numero de cores: 3 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03612511.9
 -- Numero de cores: 4 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03119311.9
 -- Numero de cores: 5 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03015711.9
 -- Numero de cores: 6 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02749811.9
 -- Numero de cores: 7 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03637311.9
```

```
 -- Numero de cores: 8 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02261611.9
 -- Numero de cores: 9 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02050911.9
 -- Numero de cores: 10 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02085111.9
 -- Numero de cores: 11 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03010011.9
 -- Numero de cores: 12 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02674511.9
 -- Numero de cores: 13 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02080711.9
 -- Numero de cores: 14 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02960311.9
 -- Numero de cores: 15 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03004611.9
 -- Numero de cores: 16 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02912711.9
 -- Numero de cores: 17 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02873111.9
 -- Numero de cores: 18 --
```

```
 -- Numero de cores: 20 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02719611.9
 -- Numero de cores: 21 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02716011.9
 -- Numero de cores: 22 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02966411.9
 -- Numero de cores: 23 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03035511.9
 -- Numero de cores: 24 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02382711.9
 -- Numero de cores: 25 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03007711.9
 -- Numero de cores: 26 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03002111.9
 -- Numero de cores: 27 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02991911.9
 -- Numero de cores: 28 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02939111.9
 -- Numero de cores: 29 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.02959611.9
 -- Numero de cores: 30 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
```

```
 -- Numero de cores: 32 --
Matriz m[0][0] = 0.000000 <--> m[7499][7499] = 0.000000
V1[0] = 0.000000 <--> V1[7499] = 749.900000
Resultado V3[0] = -1405968762.500000 <--> V3[7499] = 702843775.000018
Tiempo: 0.03059711.9
Tamaño: 50000
 -- Numero de cores: 1 --
Matriz m[0][0] = 0.000000 <--> m[49999][49999] = 0.000000
V1[0] = 0.000000 <--> V1[49999] = 4999.900000
Resultado V3[0] = -416654166750.000000 <--> V3[49999] = 208320833500.003296
Tiempo: 5.95107911.9
 -- Numero de cores: 2 --
Matriz m[0][0] = 0.000000 <--> m[49999][49999] = 0.000000
V1[0] = 0.000000 <--> V1[49999] = 4999.900000
Resultado V3[0] = -416654166750.000000 <--> V3[49999] = 208320833500.003296
Tiempo: 3.39692111.9
 -- Numero de cores: 3 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 4 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 5 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 6 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 7 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 8 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 9 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 10 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 11 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 12 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 13 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 14 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 15 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 16 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 17 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 18 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 19 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 20 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 21 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 22 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 23 --
srun: error: Unable to create step for job 98448: Job/step already completing or completed
 -- Numero de cores: 24 --
slurmstepd: error: *** JOB 98448 ON atcgrid1 CANCELLED AT 2021-04-26T23:36:36 DUE TO TIME LIMIT ***
```

TABLA (con tiempos y ganancia) Y GRÁFICA (con ganancia):

Tabla 1. Tiempos de ejecución del código secuencial y de la versión paralela para atcgrid y para el PC personal

	atcgrid1, atcgrid2 o atcgrid3				atcgrid4				PC			
	Tamaño= entre 5000 y 10000		Tamaño= entre 10000 y 100000		Tamaño= entre 5000 y 10000		Tamaño= entre 10000 y 100000		Tamaño= entre 5000 y 10000		Tamaño= entre 10000 y 100000	
Nº de núcleos (p)	T(p)	S(p)	T(p)	S(p)	T(p)	S(p)	T(p)	S(p)	T(p)	S(p)	T(p)	S(p)
Código Secuencial		----		----		----		----		----		----
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
32												

COMENTARIOS SOBRE LOS RESULTADOS: