

Arquitectura de Computadores (AC)

Cuaderno de prácticas.

Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Pablo Olivares Martínez

Grupo de prácticas y profesor de prácticas: María Isabel García Arenas

Fecha de entrega: 26/04/2021

Fecha evaluación en clase: 29/04/2021

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: El problema nos indica que no se ha especificado el ámbito de la variable `n` en el `parallel for`. Al establecer que no haya ningún ámbito por defecto, es necesario especificar el ámbito de cada variable que se use en el `parallel`. Para solucionarlo, simplemente incluimos la variable en `shared` para indicar que pertenece a dicho ámbito.

CAPTURA CÓDIGO FUENTE: `shared-clauseModificado.c`

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

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

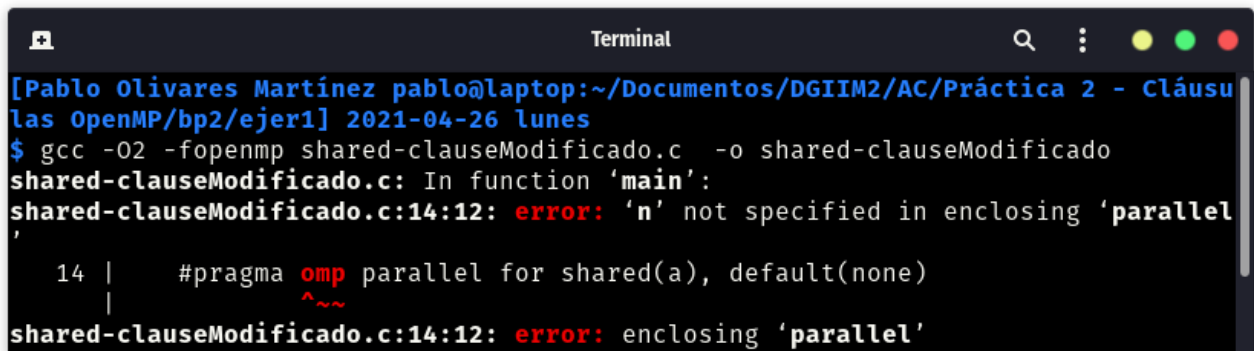
    for (i=0; i<n; i++)
        a[i] = i+1;

    #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]);
}
```

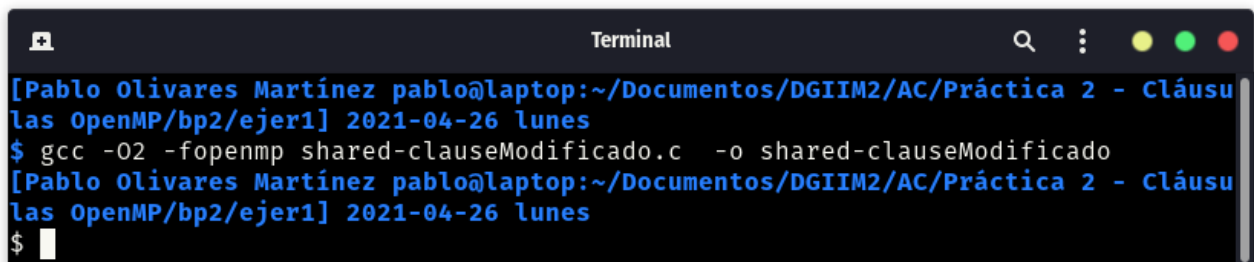
CAPTURAS DE PANTALLA:



```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer1] 2021-04-26 lunes
$ gcc -O2 -fopenmp shared-clauseModificado.c -o shared-clauseModificado
shared-clauseModificado.c: In function 'main':
shared-clauseModificado.c:14:12: error: 'n' not specified in enclosing 'parallel'
   14 |     #pragma omp parallel for shared(a), default(none)
      |            ^~~
shared-clauseModificado.c:14:12: error: enclosing 'parallel'

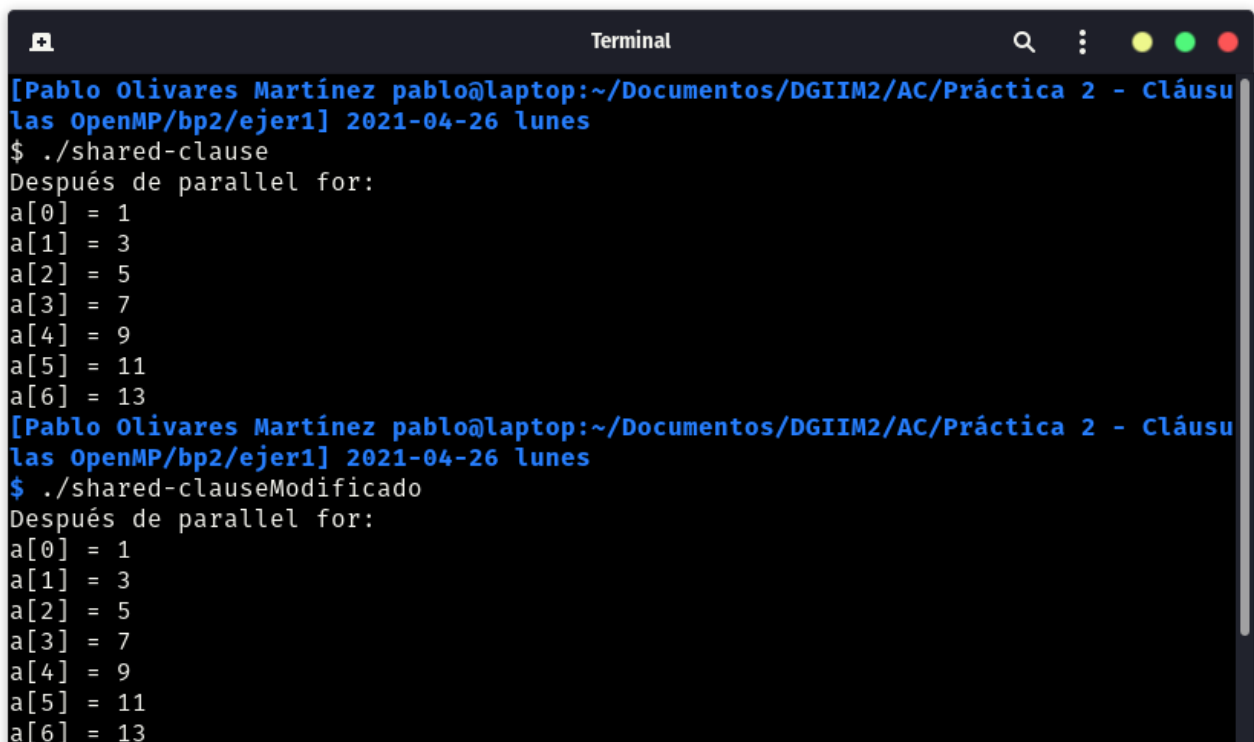
```



```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer1] 2021-04-26 lunes
$ gcc -O2 -fopenmp shared-clauseModificado.c -o shared-clauseModificado
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer1] 2021-04-26 lunes
$

```



```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer1] 2021-04-26 lunes
$ ./shared-clause
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
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer1] 2021-04-26 lunes
$ ./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

```

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: Lo que podemos observar es que, añadiendo el printf fuera del parallel, el resultado final es 0, el valor predeterminado. Sin embargo, la suma es correcta, ya que cada hilo utiliza su propia variable suma privada para realizar las operaciones y devuelve su resultado.

CAPTURA CÓDIGO FUENTE: private-clauseModificado_a.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 = 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 FUERA DEL PARALLEL: thread %d suma= %d", omp_get_thread_num(), suma);
    printf("\n");

    return 0;
}
```

CAPTURAS DE PANTALLA:

```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer2] 2021-04-26 lunes
$ ./private-clause
thread 0 suma a[0] / thread 4 suma a[4] / thread 3 suma a[3] / thread 5 suma a[5] / thread 2 suma a[2] / thread 6 suma a[6] / thread 1 suma a[1] /
* thread 0 suma= 0
* thread 4 suma= 4
* thread 6 suma= 6
* thread 2 suma= 2
* thread 1 suma= 1
* thread 3 suma= 3
* thread 7 suma= 0
* thread 5 suma= 5

```

```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer2] 2021-04-26 lunes
$ gcc -O2 -fopenmp private-clauseModificado_a.c -o private-clauseModificado_a
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer2] 2021-04-26 lunes
$ ./private-clauseModificado_a
thread 0 suma a[0] / thread 3 suma a[3] / thread 4 suma a[4] / thread 1 suma a[1] / thread 5 suma a[5] / thread 6 suma a[6] / thread 2 suma a[2] /
* thread 6 suma= 6
* thread 4 suma= 4
* thread 0 suma= 0
* thread 7 suma= 0
* thread 3 suma= 3
* thread 2 suma= 2
* thread 1 suma= 1
* thread 5 suma= 5
FUERA DEL PARALLEL: thread 0 suma= 0

```

(b) RESPUESTA: Por otro lado, el caso b es distinto. A diferencia del caso a, aquí se declara el valor de la variable suma fuera del parallel. Por tanto, al crear cada hebra su propia instancia de la variable suma y no inicializarla, estos se inicializan con basura. Sin embargo, al salir del parallel obtendremos que de resultado saldrá el valor por defecto, es decir, 0.

CAPTURA CÓDIGO FUENTE: private-clauseModificado_b.c

```

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

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

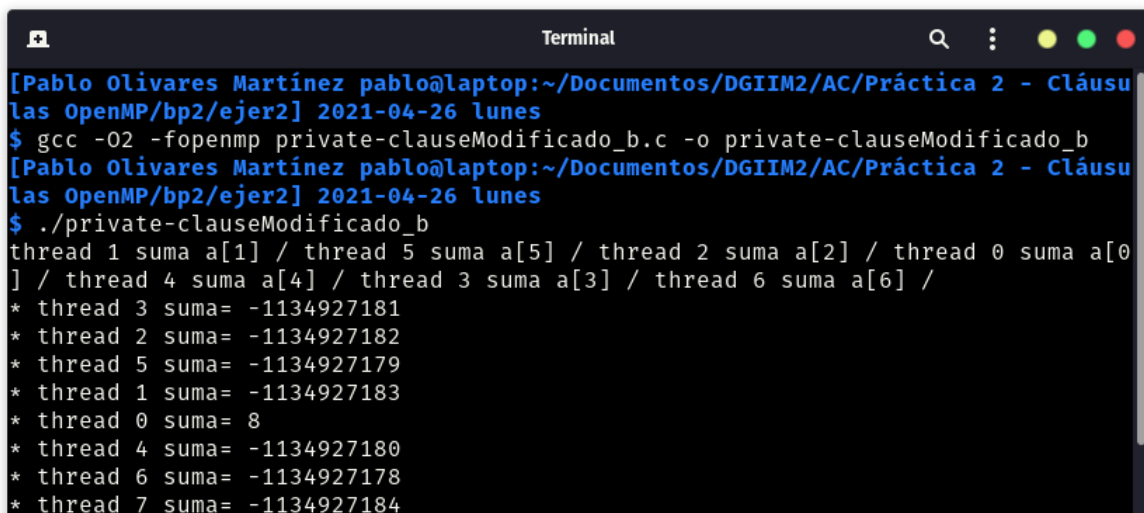
    for (i = 0; i < n; i++)
        a[i] = i;

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

    printf("\n");

    return 0;
}

```

CAPTURAS DE PANTALLA:


```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer2] 2021-04-26 lunes
$ gcc -O2 -fopenmp private-clauseModificado_b.c -o private-clauseModificado_b
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer2] 2021-04-26 lunes
$ ./private-clauseModificado_b
thread 1 suma a[1] / thread 5 suma a[5] / thread 2 suma a[2] / thread 0 suma a[0]
/ thread 4 suma a[4] / thread 3 suma a[3] / thread 6 suma a[6] /
* thread 3 suma= -1134927181
* thread 2 suma= -1134927182
* thread 5 suma= -1134927179
* thread 1 suma= -1134927183
* thread 0 suma= 8
* thread 4 suma= -1134927180
* thread 6 suma= -1134927178
* thread 7 suma= -1134927184

```

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

RESPUESTA: Lo que sucede es que la variable `suma` da 6 da igual la hebra, mientras que da diferente valor según la ejecución. Esto se debe a que, al dejar de ser privada la variable, solamente se guarda una de ellas, puesto que se van sobrescribiendo.

CAPTURA 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;

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

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

    printf("\n");

    return 0;
}
```


CAPTURAS DE PANTALLA:

```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer3] 2021-04-26 lunes
$ gcc -O2 -fopenmp -o private-clauseModificado3 private-clauseModificado3.c
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer3] 2021-04-26 lunes
$ ./private-clauseModificado3
thread 0 suma a[0] / thread 4 suma a[4] / thread 3 suma a[3] / thread 5 suma a[5] / thread 2 suma a[2] / thread 1 suma a[1] / thread 6 suma a[6] /
* thread 3 suma= 6
* thread 0 suma= 6
* thread 4 suma= 6
* thread 2 suma= 6
* thread 5 suma= 6
* thread 7 suma= 6
* thread 6 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`. **(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: Este código realiza una suma de manera similar al anterior, con la diferencia de que `firstprivate` introduce el valor de suma de fuera del contexto paralelo a éste y `lastprivate` va a exportar el valor obtenido en la última suma de dentro del `parallel` a la variable fuera de éste contexto. Es por ello que obtenemos valores distintos en cada ejecución.

CAPTURAS DE PANTALLA:

```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer4] 2021-04-26 lunes
$ gcc -O2 -fopenmp firstlastprivate-clauseModificado.c -o firstlastprivate-clauseModificado
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer4] 2021-04-26 lunes
$ ./firstlastprivate-clauseModificado
thread 4 suma a[4] suma=4
thread 3 suma a[3] suma=3
thread 5 suma a[5] suma=5
thread 1 suma a[1] suma=1
thread 2 suma a[2] suma=2
thread 6 suma a[6] suma=6
thread 0 suma a[0] suma=0

Fuera de la construcción parallel suma=6
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer4] 2021-04-26 lunes
$ ./firstlastprivate-clauseModificado
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 4 suma a[6] suma=6
thread 7 suma a[9] suma=9
thread 3 suma a[5] suma=5
thread 2 suma a[4] suma=4
thread 6 suma a[8] suma=8
thread 5 suma a[7] suma=7
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5

Fuera de la construcción parallel suma=9

```

(b) RESPUESTA: Aunque en este caso los resultados coincidan, no siempre tienen por qué hacerlo, ya que éste resultado final depende de las condiciones iniciales a la hora de ejecutar el código, como el número de hebras, el reparto dado, etc.

CAPTURAS DE PANTALLA:

```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer4] 2021-04-26 lunes
$ ./firstlastprivate-clause
thread 1 suma a[1] suma=1
thread 5 suma a[5] suma=5
thread 3 suma a[3] suma=3
thread 4 suma a[4] suma=4
thread 0 suma a[0] suma=0
thread 2 suma a[2] suma=2
thread 6 suma a[6] suma=6

Fuera de la construcción parallel suma=6
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer4] 2021-04-26 lunes
$ ./firstlastprivate-clause
thread 3 suma a[3] suma=3
thread 6 suma a[6] suma=6
thread 0 suma a[0] suma=0
thread 4 suma a[4] suma=4
thread 5 suma a[5] suma=5
thread 1 suma a[1] suma=1
thread 2 suma a[2] suma=2

Fuera de la construcción parallel suma=6
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer4] 2021-04-26 lunes
$ ./firstlastprivate-clause
thread 4 suma a[4] suma=4
thread 1 suma a[1] suma=1
thread 2 suma a[2] suma=2
thread 5 suma a[5] suma=5
thread 6 suma a[6] suma=6
thread 3 suma a[3] suma=3
thread 0 suma a[0] suma=0

Fuera de la construcción parallel suma=6

```

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: Cuando se elimina la cláusula `copyprivate`, lo que sucede es que la hebra que ha ejecutado la directiva `single` ha inicializado las componentes del vector que se le han asignado, mientras el resto se mantienen a 0.

CAPTURA CÓDIGO FUENTE: `copyprivate-clauseModificado.c`


```
#include <omp.h>
#include <stdio.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 inicializacion 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("Despues de la region parallel:\n");

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

    printf("\n");
}
```

CAPTURAS DE PANTALLA:

```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer5] 2021-04-26 lunes
$ gcc -O2 -fopenmp copyprivate-clauseModificado.c -o copyprivate-clauseModificad
o
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer5] 2021-04-26 lunes
$ ./copyprivate-clause

Introduce valor de inicialización a: 8

Single ejecutada por el thread 3
Después de la región parallel:
b[0] = 8      b[1] = 8      b[2] = 8      b[3] = 8      b[4] = 8      b
[5] = 8 b[6] = 8      b[7] = 8      b[8] = 8
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer5] 2021-04-26 lunes
$ ./copyprivate-clauseModificado

Introduce valor de inicialización a: 8

Single ejecutada por el thread 5
Después de la región parallel:
b[0] = 22038   b[1] = 22038   b[2] = 0      b[3] = 0      b[4] = 0      b
[5] = 0 b[6] = 8      b[7] = 0      b[8] = 0
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer5] 2021-04-26 lunes
$ ./copyprivate-clause

Introduce valor de inicialización a: 12

Single ejecutada por el thread 3
Después de la región parallel:
b[0] = 12      b[1] = 12      b[2] = 12      b[3] = 12      b[4] = 12      b
[5] = 12      b[6] = 12      b[7] = 12      b[8] = 12
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer5] 2021-04-26 lunes
$ ./copyprivate-clauseModificado

Introduce valor de inicialización a: 12

Single ejecutada por el thread 5
Después de la región parallel:
b[0] = 22022   b[1] = 22022   b[2] = 0      b[3] = 0      b[4] = 0      b
[5] = 0 b[6] = 12      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: Hace lo mismo que el programa con `suma=0` pero como usa la cláusula `reduction`, la cual mantiene el valor inicial, le suma 10 en este caso.

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado.c`

```

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

int main(int argc, char **argv) {
    int i, n = 20, a[n], suma = 10;

    if (argc < 2) {
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]);
    if (n > 20) {
        n = 20;
        printf("n=%d", n);
    }

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

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

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

```

CAPTURAS DE PANTALLA:

```

Terminal
^[[A[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer6] 2021-04-26 lunes
$ gcc -O2 -fopenmp reduction-clauseModificado.c -o reduction-clauseModificado
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer6] 2021-04-26 lunes
$ ./reduction-clause 4
Tras 'parallel' suma=6
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer6] 2021-04-26 lunes
$ ./reduction-clauseModificado 4
Tras 'parallel' suma=16
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer6] 2021-04-26 lunes
$ ./reduction-clause 8
Tras 'parallel' suma=28
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer6] 2021-04-26 lunes
$ ./reduction-clauseModificado 8
Tras 'parallel' suma=38

```

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).

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado7.c`

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

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

    if (argc < 2) {
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]);
    if (n > 20) {
        n = 20;
        printf("n=%d", n);
    }

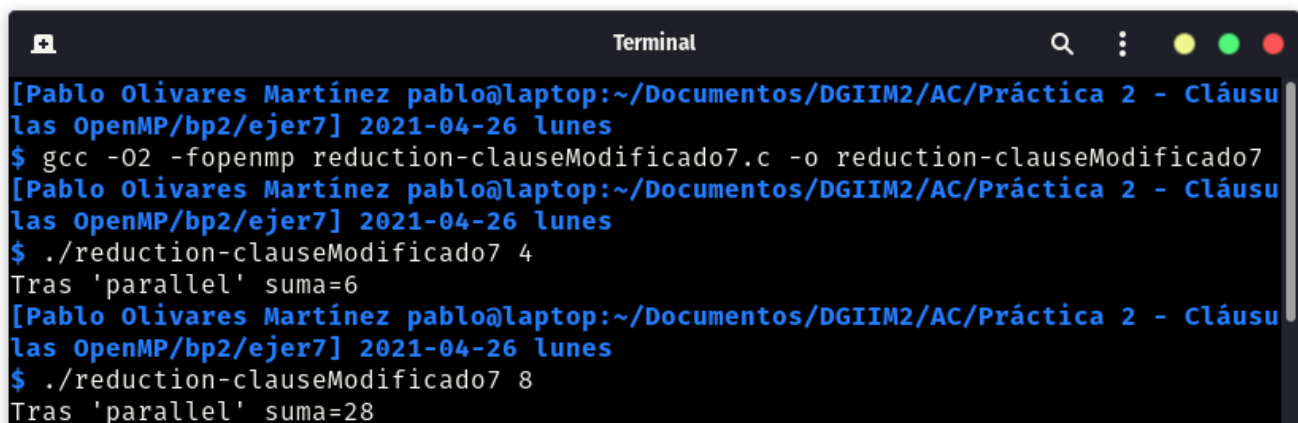
    for (i = 0; i < n; i++) a[i] = i;

    #pragma omp parallel
    {
        int suma_parallel = 0;

        #pragma omp for
        for (i = 0; i < n; i++) suma_parallel += a[i];

        #pragma omp atomic
        suma += suma_parallel;
    }
    printf("Tras 'parallel' suma=%d\n", suma);
}
```

CAPTURAS DE PANTALLA:



```
Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer7] 2021-04-26 lunes
$ gcc -O2 -fopenmp reduction-clauseModificado7.c -o reduction-clauseModificado7
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer7] 2021-04-26 lunes
$ ./reduction-clauseModificado7 4
Tras 'parallel' suma=6
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer7] 2021-04-26 lunes
$ ./reduction-clauseModificado7 8
Tras 'parallel' suma=28
```

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):

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

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).

CAPTURA CÓDIGO FUENTE: pmv-secuencial.c

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #ifdef _OPENMP
5      #include <omp.h>
6  #else
7      #define omp_get_thread_num() 0
8      #define omp_get_num_threads() 1
9  #endif
10
11  //Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de los
12  //dos defines siguientes puede estar descomentado):
13  // #define VECTOR_GLOBAL // descomentar para que los vectores sean variables ...
14  // globales (su longitud no estará limitada por el ...
15  // tamaño de la pila del programa)
16  #define VECTOR_DYNAMIC // descomentar para que los vectores sean variables ...
17  // dinámicas (memoria reutilizable durante la ejecución)
18
19  #ifdef VECTOR_GLOBAL
20      #define MAX 1024 // = 2^10
21      double v1[MAX], v2[MAX], M[MAX][MAX];
22  #endif
23
24  int main(int argc, char** argv) {
25      int i, j;
26      double t1, t2, t3;
27
28      if (argc < 2) {
29          printf("Falta tamaño de matriz y vector\n");
30          exit(-1);
31      }
32
33      unsigned int N = atoi(argv[1]);
34
35      #ifdef VECTOR_GLOBAL
36          if (N > MAX) N = MAX;
37      #endif

```



```

40     #ifdef VECTOR_DYNAMIC
41         double *v1, *v2, **M;
42         v1 = (double *)malloc(N * sizeof(double));
43         v2 = (double *)malloc(N * sizeof(double));
44         M = (double **)malloc(N * sizeof(double *));
45
46         // Reservamos memoria ahora para las componentes de la matriz
47         for (i = 0; i < N; i++) {
48             M[i] = (double *)malloc(N * sizeof(double));
49         }
50         if ((v1 == NULL) || (v2 == NULL) || (v2 == NULL)) {
51             printf("No hay suficiente espacio para los vectores \n");
52             exit(-2);
53         }
54     #endif
55
56     // Inicializamos la matriz y los vectores
57     for (i=0; i<N;i++){
58         v1[i] = i;
59         v2[i] = 0;
60         for(j=0;j<N;j++)
61             M[i][j] = i+j;
62     }
63
64     // Inicializamos la primera variable de tiempo
65     t1 = omp_get_wtime();
66
67     // Calculamos el producto
68     for (i=0; i<N;i++)
69         for(j=0;j<N;j++)
70             v2[i] += M[i][j] * v1[j];
71
72
73     t2 = omp_get_wtime();
74     t3 = t2 - t1;
75
76     // Para tamaños pequeños (hasta N=10), imprimimos todas las componentes
77     // y el tiempo de ejecución
78     if (N <= 10) {
79         printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n", N, t3);
80
81         for (i = 0; i < N; i++) {
82             printf("v2[%i] = %f\n", i, v2[i]);
83         }
84     }

```



```

85
86     // Para tamaños superiores, imprimimos el tiempo de ejecución y la primera y últi
87     else {
88         printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n Primera componente: %
89             v2[0], v2[N - 1]);
90     }
91
92     #ifdef VECTOR_DYNAMIC
93         free(v1);
94         free(v2);
95         for (i = 0; i < N; i++) {
96             free(M[i]);
97         }
98         free(M);
99     #endif
100
101     return 0;
102 }
103

```

CAPTURAS DE PANTALLA:

```

Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer8] 2021-04-27 martes
$ gcc -fopenmp -O2 pmv-secuencial.c -o pmv-secuencial-global
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer8] 2021-04-27 martes
$ gcc -fopenmp -O2 pmv-secuencial.c -o pmv-secuencial-dynamic

```

```

e1estudiante21@atcgrid:~/bp2/ejer8
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer8] 2021-04-27 martes
$ srun pmv-secuencial-global 8
Tamaño vectores: 8
Tiempo de ejecución: 0.000000
v2[0] = 140.000000
v2[1] = 168.000000
v2[2] = 196.000000
v2[3] = 224.000000
v2[4] = 252.000000
v2[5] = 280.000000
v2[6] = 308.000000
v2[7] = 336.000000
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer8] 2021-04-27 martes
$ srun pmv-secuencial-global 100
Tamaño vectores: 100
Tiempo de ejecución: 0.000020
Primera componente: 328350.000000
Última componente: 818400.000000
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer8] 2021-04-27 martes

```

```

e1estudiante21@atcgrid:~/bp2/ejer8
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer8] 2021-04-27 martes
$ srun pmv-secuencial-dynamic 8
Tamaño vectores: 8
Tiempo de ejecución: 0.000000
v2[0] = 140.000000
v2[1] = 168.000000
v2[2] = 196.000000
v2[3] = 224.000000
v2[4] = 252.000000
v2[5] = 280.000000
v2[6] = 308.000000
v2[7] = 336.000000
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer8] 2021-04-27 martes
$ srun pmv-secuencial-dynamic 100
Tamaño vectores: 100
Tiempo de ejecución: 0.000020
Primera componente: 328350.000000
Última componente: 818400.000000
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer8] 2021-04-27 martes

```

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).

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

int main(int argc, char **argv) {
    int i, j;
    double t1, total;

    if(argc < 2){
        fprintf(stderr, "Falta tamaño de la matriz (debe de ser cuadrada)\n");
        exit(-1);
    }

    // Tamaño de la matriz pasado por parámetro
    unsigned int tam = atoi(argv[1]);

    if(tam < 2){
        fprintf(stderr, "El tamaño no puede ser menor a 2");
        exit(-1);
    }

    // Declaramos con memoria dinámica
    double *v1, *v2, **m;

    v1 = (double*) malloc(tam*sizeof(double));
    v2 = (double*) malloc(tam*sizeof(double));
    m = (double**) malloc(tam*sizeof(double*));

    // Reservamos memoria ahora para las componentes de la matriz
    #pragma omp parallel for
    for(i=0; i<tam; i++)
        m[i] = (double*) malloc(tam*sizeof(double));

    #pragma omp parallel private(i)
    {
        // Inicializamos la matriz y los vectores
        #pragma omp for
        for(i=0; i<tam; i++)
            v1[i] = 1;
            v2[i] = 0;

        #pragma omp for private(j) // Para paralelizar el for de dentro
        for(i=0; i<tam; i++){
            for(j=0; j<tam; j++){
                m[i][j] = 2;
            }
        }
    }
}
```

```

// Inicializamos la primera variable de tiempo
#pragma omp single
{
    t1 = omp_get_wtime();
}

// Calculamos el producto
#pragma omp for private(j) // Para paralelizar el for de dentro
for(i=0; i<tam; i++){
    for(j=0; j<tam; j++){
        v2[i] = v2[i] + (m[i][j]*v1[j]);
    }
}

// Obtenemos el tiempo total transcurrido
#pragma omp single
{
    total = omp_get_wtime() - t1;
}

// Para tamaños pequeños (hasta tam=11), imprimimos todas las componentes del vector
// y el tiempo de ejecución
if(tam <= 11){
    printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n", tam, total);

    for(i=0; i<tam; i++){
        printf("v2[%i] = %f\n", i, v2[i]);
    }
}

// Para tamaños superiores, imprimimos el tiempo de ejecución y la primera y última
else{
    printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n Primera componente: %f\n", tam, total, v2[0]);
}

// Liberamos memoria
free(v1);
free(v2);

for(i=0; i<tam; i++){
    free(m[i]);
}

free(m);

return 0;

```

CAPTURA CÓDIGO FUENTE: pmv-OpenMP-b.c

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

int main(int argc, char **argv) {
    int i, j;
    double t1, total;

    if(argc < 2){
        fprintf(stderr, "Falta tamaño de la matriz (debe de ser cuadrada)\n");
        exit(-1);
    }

    // Tamaño de la matriz pasado por parámetro
    unsigned int tam = atoi(argv[1]);

    if(tam < 2){
        fprintf(stderr, "El tamaño no puede ser menor a 2");
        exit(-1);
    }

    // Declaramos con memoria dinámica
    double *v1, *v2, **m;

    v1 = (double*) malloc(tam*sizeof(double));
    v2 = (double*) malloc(tam*sizeof(double));
    m = (double**) malloc(tam*sizeof(double*));

    // Reservamos memoria ahora para las componentes de la matriz
    #pragma omp parallel for
    for(i=0; i<tam; i++)
        m[i] = (double*) malloc(tam*sizeof(double));

    #pragma omp parallel private(i)
    {
        // Inicializamos la matriz y los vectores
        #pragma omp for
        for(i=0; i<tam; i++)
            v1[i] = 1;
            v2[i] = 0;

        #pragma omp for private(j) // Para paralelizar el for de dentro
        for(i=0; i<tam; i++){
            for(j=0; j<tam; j++){
                m[i][j] = 2;
            }
        }
    }
}
```



```

#pragma omp single
{
    t1 = omp_get_wtime();
}

// Calculamos el producto
for(i=0; i<tam; i++){
    double producto_local = 0;

    #pragma omp for
    for(j=0; j<tam; j++){
        producto_local = producto_local + (m[i][j]*v1[j]);
    }

    #pragma omp critical
    v2[i] += producto_local;
}

// Obtenemos el tiempo total transcurrido
#pragma omp single
{
    total = omp_get_wtime() - t1;
}

// Para tamaños pequeños (hasta tam=11), imprimimos todas las componentes del vector resultado
// y el tiempo de ejecución
if(tam <= 11){
    printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n", tam, total);

    for(i=0; i<tam; i++){
        printf("v2[%i] = %f\n", i, v2[i]);
    }
}

// Para tamaños superiores, imprimimos el tiempo de ejecución y la primera y última componente
else{
    printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n Primera componente: %f\n Última componente: %f\n", tam, total, v2[0], v2[tam-1]);
}

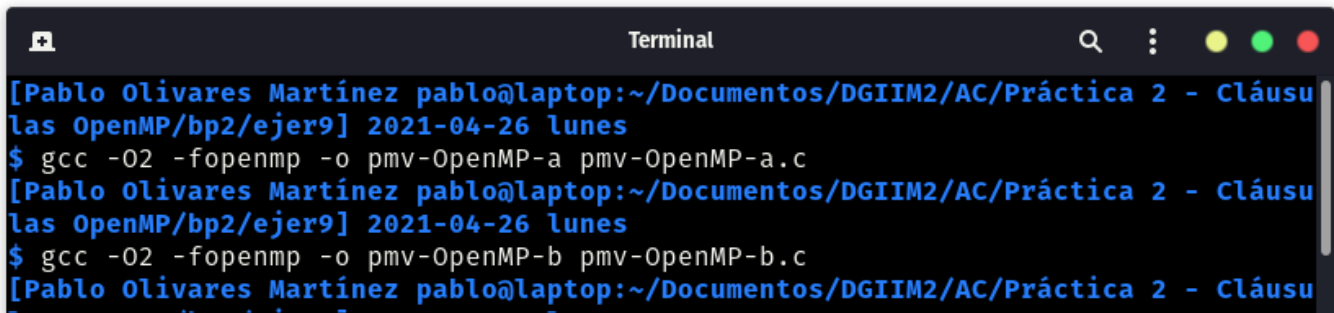
// Liberamos memoria
free(v1);
free(v2);

for(i=0; i<tam; i++){
    free(m[i]);
}

```

RESPUESTA: No estaba muy seguro de cómo paralelizar bucles anidados, así que acudí a StackOverflow para encontrar la solución. Finalmente decidí usar `private` sobre la variable de bucle anidado para solventar mi problema. (Fuente: <https://stackoverflow.com/questions/50909121/nested-loop-openmp-parallelizing-private-or-public-index>)

CAPTURAS DE PANTALLA:



```
Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer9] 2021-04-26 lunes
$ gcc -O2 -fopenmp -o pmv-OpenMP-a pmv-OpenMP-a.c
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
las OpenMP/bp2/ejer9] 2021-04-26 lunes
$ gcc -O2 -fopenmp -o pmv-OpenMP-b pmv-OpenMP-b.c
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusu
```



```
e1estudiante21@atcgrid:~/bp2/ejer9
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer9] 2021-04-26 lunes
$ srun pmv-OpenMP-a 10000
Tamaño vectores: 10000
Tiempo de ejecución: 0.140583
Primera componente: 20000.000000
Última componente: 20000.000000
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer9] 2021-04-26 lunes
$ srun pmv-OpenMP-b 10000
Tamaño vectores: 10000
Tiempo de ejecución: 0.139024
Primera componente: 20000.000000
```

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

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

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

```

int main(int argc, char **argv) {
    int i, j;
    double t1, total, producto_local = 0;

    if(argc < 2){
        fprintf(stderr, "Falta tamaño de la matriz (debe de ser cuadrada)\n");
        exit(-1);
    }

    // Tamaño de la matriz pasado por parámetro
    unsigned int tam = atoi(argv[1]);

    if(tam < 2){
        fprintf(stderr, "El tamaño no puede ser menor a 2");
        exit(-1);
    }

    // Declaramos con memoria dinámica
    double *v1, *v2, **m;

    v1 = (double*) malloc(tam*sizeof(double));
    v2 = (double*) malloc(tam*sizeof(double));
    m = (double**) malloc(tam*sizeof(double*));

    // Reservamos memoria ahora para las componentes de la matriz
    #pragma omp parallel for
    for(i=0; i<tam; i++)
        m[i] = (double*) malloc(tam*sizeof(double));

    #pragma omp parallel private(i)
    {
        // Inicializamos la matriz y los vectores
        #pragma omp for
        for(i=0; i<tam; i++)
            v1[i] = 1;
            v2[i] = 0;

        #pragma omp for private(j) // Para paralelizar el for de dentro
        for(i=0; i<tam; i++){
            for(j=0; j<tam; j++){
                m[i][j] = 2;
            }
        }
    }

    // Inicializamos la primera variable de tiempo

```

```

#pragma omp single
{
    t1 = omp_get_wtime();
}

// Calculamos el producto
for(i=0; i<tam; i++){

    #pragma omp for reduction(+:producto_local)
    for(j=0; j<tam; j++){
        producto_local = producto_local + (m[i][j]*v1[j]);
    }

    #pragma omp single
    {
        v2[i] = producto_local;
        producto_local = 0;
    }
}

// Obtenemos el tiempo total transcurrido
#pragma omp single
{
    total = omp_get_wtime() - t1;
}

// Para tamaños pequeños (hasta tam=11), imprimimos todas las componentes del vector v2
// y el tiempo de ejecución
if(tam <= 11){
    printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n", tam, total);

    for(i=0; i<tam; i++){
        printf("v2[%i] = %f\n", i, v2[i]);
    }
}

// Para tamaños superiores, imprimimos el tiempo de ejecución y la primera y última componente
else{
    printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n Primera componente: %f\n Última componente: %f\n", tam, total, v2[0], v2[tam-1]);
}

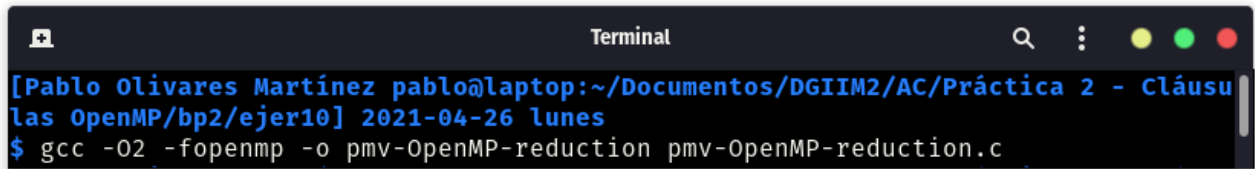
// Liberamos memoria
free(v1);
free(v2);

for(i=0; i<tam; i++){

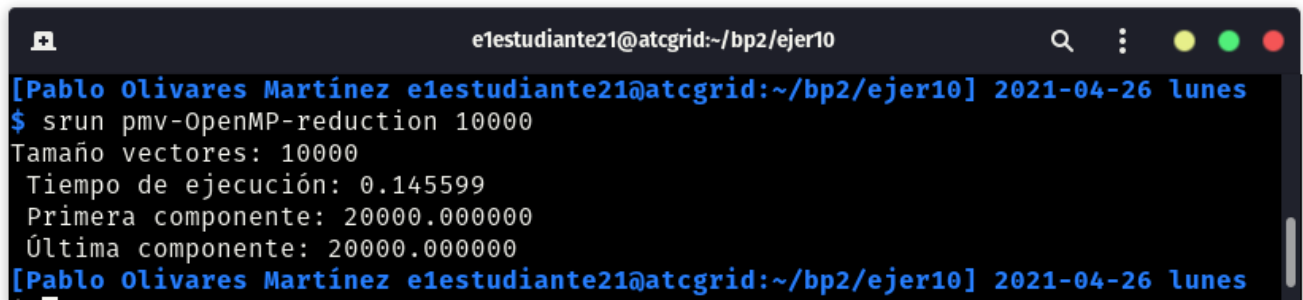
```

RESPUESTA: Tuve un problema con la definición de `producto_local`, ya que era privada. Para solucionarlo, saqué la variable fuera del `for` para calcular el producto de la matriz.

CAPTURAS DE PANTALLA:



```
Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer10] 2021-04-26 lunes
$ gcc -O2 -fopenmp -o pmv-OpenMP-reduction pmv-OpenMP-reduction.c
```



```
e1estudiante21@atcgrid:~/bp2/ejer10
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer10] 2021-04-26 lunes
$ srun pmv-OpenMP-reduction 10000
Tamaño vectores: 10000
Tiempo de ejecución: 0.145599
Primera componente: 20000.000000
Última componente: 20000.000000
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer10] 2021-04-26 lunes
```

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):

NÚMERO DE DATOS: 7500												
THREAD/BIN	PC				ATCGRID				ATCGRID4			
	SECUENCIA	OMP-A	OMP-B	REDUCTION	SECUENCIA	OMP-A	OMP-B	REDUCTION	SECUENCIA	OMP-A	OMP-B	REDUCTION
1	0,102084	0,10817	0,10911	0,112233	0,080581	0,08251	0,085416	0,089034	0,107037	0,075232	0,076879	0,079567
2	0,102121	0,053661	0,056996	0,062239	0,080554	0,079027	0,079171	0,084129	0,074943	0,04064	0,044991	0,048103
3	0,102893	0,035994	0,039831	0,044955	0,080619	0,079152	0,143964	0,188061	0,073429	0,042811	0,097123	0,140308
4	0,102172	0,027167	0,032821	0,036544	0,08057	0,078348	0,139736	0,200545	0,073552	0,044164	0,087646	0,123055
5	0,102259	0,023373	0,029276	0,033581	0,083544	0,079258	0,175708	0,238418	0,073954	0,041652	0,126931	0,204667
6	0,102305	0,020341	0,028914	0,030787	0,087244	0,080547	0,168208	0,237797	0,074519	0,042127	0,12257	0,200784
7	0,102309	0,019141	0,027705	0,029416	0,080539	0,079454	0,195494	0,30111	0,073797	0,045925	0,157293	0,272268
8	0,102569	0,018978	0,027872	0,029816	0,080528	0,07942	0,2031	0,325495	0,073807	0,041203	0,162397	0,2894
9	0,102334	0,026805	0,03289	0,0385129	0,080632	0,078947	0,226453	0,375986	0,074181	0,043018	0,202634	0,362015
10	0,102118	0,026068	0,220816	0,417834	0,080623	0,079123	0,238646	0,406398	0,072962	0,041561	0,201307	0,378625
11	0,10235	0,023755	0,238812	0,4569	0,080566	0,07922	0,266905	0,44855	0,074193	0,044485	0,227484	0,455021
12	0,102567	0,022259	0,255458	0,482732	0,08062	0,07913	0,274652	0,479605	0,074446	0,042187	0,237678	0,469189
13	0,10229	0,022063	0,280505	0,574354	0,080532	0,079595	0,305913	0,512642	0,07493	0,041437	0,272838	0,536602
14	0,102123	0,020439	0,289072	0,544115	0,080573	0,07901	0,318413	0,551852	0,074928	0,040733	0,308575	0,529877
15	0,10218	0,019115	0,303435	0,584233	0,080579	0,079064	0,332692	0,589239	0,074912	0,041803	0,357291	0,566818
16	0,10214	0,019372	0,317952	0,632277	0,080564	0,079043	0,356262	0,620168	0,074939	0,041452	0,339731	0,634281
32	0,102216	0,021103	0,656749	1,277834	0,080587	0,080435	0,646215	1,163651	0,07495	0,041311	0,652203	1,208436
MEDIA T(p)	0,102295882	0,029870824	0,183447824	0,337351706	0,081144412	0,079487235	0,244526353	0,400745882	0,076204647	0,044220059	0,216210059	0,382295059

NÚMERO DE DATOS: 10500												
THREAD/BIN	PC				ATCGRID				ATCGRID4			
	SECUENCIA	OMP-A	OMP-B	REDUCTION	SECUENCIA	OMP-A	OMP-B	REDUCTION	SECUENCIA	OMP-A	OMP-B	REDUCTION
1	0,200944	0,210802	0,208588	0,214674	0,158063	0,16111	0,164022	0,16973	0,143056	0,147735	0,148029	0,151679
2	0,200791	0,105575	0,109014	0,115963	0,158478	0,155639	0,152673	0,15982	0,143916	0,082225	0,085755	0,090663
3	0,201142	0,070242	0,075313	0,082628	0,158103	0,154379	0,262552	0,323216	0,143441	0,084103	0,166005	0,224787
4	0,200807	0,053288	0,06054	0,066501	0,160434	0,156776	0,257252	0,339136	0,146578	0,085682	0,14157	0,204984
5	0,200805	0,045729	0,053628	0,061113	0,167486	0,155284	0,271677	0,431139	0,144641	0,082453	0,204365	0,314963
6	0,201384	0,039836	0,049176	0,057982	0,163654	0,158617	0,276992	0,379277	0,144657	0,081024	0,195958	0,30208
7	0,200917	0,038141	0,048626	0,051397	0,158802	0,154855	0,323674	0,468009	0,143235	0,083028	0,245273	0,404409
8	0,201081	0,036471	0,049513	0,051326	0,158077	0,155955	0,318377	0,501345	0,146543	0,081297	0,256132	0,46072
9	0,201271	0,044463	0,300296	0,554817	0,158025	0,156455	0,36282	0,566705	0,147203	0,083366	0,305066	0,542994
10	0,201123	0,049582	0,321823	0,603124	0,158279	0,1562	0,382129	0,609238	0,1447	0,0815	0,307624	0,616101
11	0,200832	0,038566	0,347929	0,651023	0,157958	0,155365	0,420774	0,677431	0,147063	0,082512	0,35747	0,680221
12	0,20153	0,038964	0,36959	0,695404	0,158023	0,157834	0,431381	0,714948	0,146107	0,082624	0,416868	0,714849
13	0,201559	0,039058	0,4098	0,727792	0,157955	0,155368	0,46778	0,765095	0,145168	0,08016	0,472907	0,833246
14	0,200657	0,039372	0,405305	0,76718	0,158132	0,157725	0,482939	0,819907	0,147452	0,080807	0,466618	0,88034
15	0,200607	0,037991	0,434211	0,829406	0,15802	0,155952	0,510166	0,870007	0,14718	0,082796	0,514526	0,865834
16	0,200837	0,04424	0,457095	0,89121	0,157983	0,154825	0,538825	0,905664	0,145294	0,081257	0,457608	0,991701
32	0,201832	0,041409	0,93121	1,814135	0,158011	0,155585	0,943053	1,660723	0,146975	0,08219	0,923294	1,767955
MEDIA T(p)	0,201065824	0,057278176	0,272450412	0,484451471	0,159146059	0,156364059	0,386299176	0,609493529	0,145482882	0,086162294	0,333239294	0,591030941

JUSTIFICAR AHORA EN BASE AL CÓDIGO LA DIFERENCIA EN TIEMPOS: Para saber cual es el mejor código, he ejecutado todos ellos según distintos threads, y obtenido el promedio para saber cual ha sido el más rápido de media. El código más rápido ha resultado ser el OMP-A, ya que al realizar la paralización por filas, el programa pierde menos tiempo comunicando los procesos necesarios para las operaciones.

CAPTURA DE PANTALLA del script pmv-OpenmMP-script.sh (las diferentes versiones del script se encuentran en la carpeta del ejercicio)

```
#!/bin/bash
#Órdenes para el Gestor de carga de un trabajo:
#1. Asigna al trabajo un nombre
#SBATCH --job-name=tiempos_pmv_ac
#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 trabajo (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"

#Declaración de variables
X0=1
XN=16

#Función auxiliar
execute_bin()
{
    VALUE_1=7500
    VALUE_2=10500

    export OMP_NUM_THREADS=$1

    echo "PMV SECUENCIAL"
    srun ./pmv-secuencial-dynamic $VALUE_1
    srun ./pmv-secuencial-dynamic $VALUE_2
    echo

    echo "PMV OPENMP-A"
    srun ./pmv-OpenMP-a $VALUE_1
    srun ./pmv-OpenMP-a $VALUE_2
    echo

    echo "PMV OPENMP-B"
    srun ./pmv-OpenMP-b $VALUE_1
    srun ./pmv-OpenMP-b $VALUE_2
    echo

    echo "PMV OPENMP-REDUCTION"
    srun ./pmv-OpenMP-reduction $VALUE_1
    srun ./pmv-OpenMP-reduction $VALUE_2
    echo
}

#Instrucciones del script para ejecutar código:

for ((P=$X0;P<=$XN;P=P+1))
do
    echo -e "NÚMERO DE HEBRAS: $P \n"
    execute_bin $P
done

echo -e "NÚMERO DE HEBRAS: 32 \n"
execute_bin 32
```

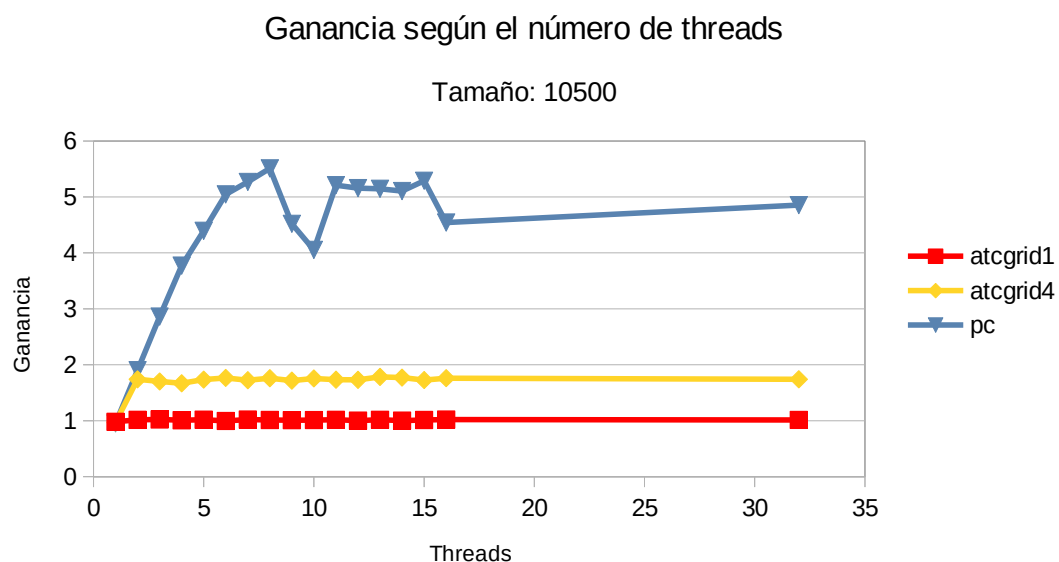
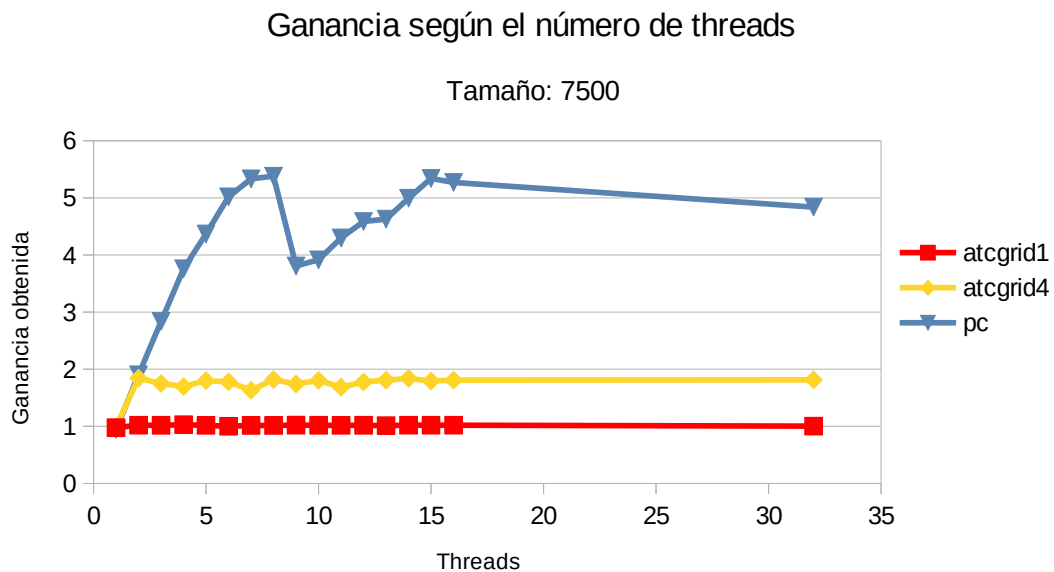

CAPTURAS DE PANTALLA (mostrar la ejecución en atcgrid – envío(s) a la cola):

```
Terminal
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer11] 2021-04-27 martes
$ ./script_pmv_PC.sh > tiempos_pmv_pc.dat
[Pablo Olivares Martínez pablo@laptop:~/Documentos/DGIIM2/AC/Práctica 2 - Cláusulas OpenMP/bp2/ejer11] 2021-04-27 martes
$
```

```
e1estudiante21@atcgrid:~/bp2/ejer11
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer11] 2021-04-27 martes
$ sbatch script_pmv_ATCGRID.sh
Submitted batch job 100203
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer11] 2021-04-27 martes
$ sbatch script_pmv_ATCGRID4.sh
Submitted batch job 100204
[Pablo Olivares Martínez e1estudiante21@atcgrid:~/bp2/ejer11] 2021-04-27 martes
$
```

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

OMP-A Tamaño n° núcleos (p)	ATCGRID1/2/3				ATCGRID4				PC			
	7500		10500		7500		10500		7500		10500	
	T(p)	S(p)	T(p)	S(p)	T(p)	S(p)	T(p)	S(p)	T(p)	S(p)	T(p)	S(p)
Secuencial	0,080581	---	0,158063	---	0,074943	---	0,143056	---	0,102084	---	0,200944	---
1	0,08251	0,976621016	0,16111	0,981087456	0,075232	0,99615855	0,147735	0,968328426	0,10817	0,943736711	0,210802	0,953235738
2	0,079027	1,019664165	0,155639	1,015574503	0,04064	1,844069882	0,082225	1,739811493	0,053661	1,902387209	0,105575	1,903329387
3	0,079152	1,018053871	0,154379	1,023863349	0,042811	1,750554764	0,084103	1,700961916	0,035994	2,836139357	0,070242	2,860738589
4	0,078348	1,028501047	0,156776	1,008209165	0,044164	1,696925097	0,085682	1,669615555	0,027167	3,757647145	0,053288	3,770905269
5	0,079258	1,016692321	0,155284	1,017896242	0,041652	1,799265341	0,082453	1,735000546	0,023373	4,367603645	0,045729	4,394235605
6	0,080547	1,000422114	0,158617	0,99650731	0,042127	1,778977853	0,081024	1,765600316	0,020341	5,018632319	0,039836	5,044281554
7	0,079454	1,014184308	0,154855	1,020716154	0,045925	1,631856287	0,083028	1,722985017	0,019141	5,333263675	0,038141	5,268451273
8	0,07942	1,014618484	0,155955	1,01351672	0,041203	1,818872412	0,081297	1,759671329	0,018978	5,379070503	0,036471	5,509692633
9	0,078947	1,02069743	0,156455	1,010277716	0,043018	1,742131201	0,083366	1,715999328	0,026805	3,808393956	0,044463	4,51935317
10	0,079123	1,018427006	0,1562	1,011927017	0,041561	1,803204928	0,0815	1,755288344	0,026068	3,916065674	0,049582	4,052761083
11	0,07922	1,017180005	0,155365	1,017365559	0,044485	1,684680229	0,082512	1,733759938	0,023755	4,297368975	0,038566	5,210392574
12	0,07913	1,018336914	0,157834	1,001450891	0,042187	1,776447721	0,082624	1,73140976	0,022259	4,586189856	0,038964	5,157170722
13	0,079595	1,012387713	0,155368	1,017345914	0,041437	1,808601009	0,08016	1,784630739	0,022063	4,626931968	0,039058	5,144759076
14	0,07901	1,019883559	0,157725	1,00214297	0,040733	1,839859573	0,080807	1,770341678	0,020439	4,994569206	0,039372	5,103728538
15	0,079064	1,019186988	0,155952	1,013536216	0,041803	1,792766069	0,082796	1,727812938	0,019115	5,340517918	0,037991	5,289252718
16	0,079043	1,019457763	0,154825	1,020913935	0,041452	1,807946541	0,081257	1,760537554	0,019372	5,269667561	0,04424	4,542133816
32	0,080435	1,00181513	0,15585	1,014199551	0,041311	1,814117305	0,08219	1,740552379	0,021103	4,837416481	0,041409	4,85266488



COMENTARIOS SOBRE LOS RESULTADOS:

Según podemos ver en las gráficas, la paralelización en atcgrid mantiene una ganancia en prestaciones prácticamente constante, mejorando el código secuencial. Por otro lado, en el PC vemos que se produce una mejora aún más drástica, pero con más variaciones. Esto se debe a que mi PC tiene 8 cores lógicos, por lo que cuando los supera y no es múltiplo de éste valor, la ganancia no es tan buena como hasta 8 threads.