

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): Javier Ramirez Pulido

Grupo de prácticas y profesor de prácticas: C2 Jorge Sanchez Garrido

Fecha de entrega: 03/04/2020

Fecha evaluación en clase:

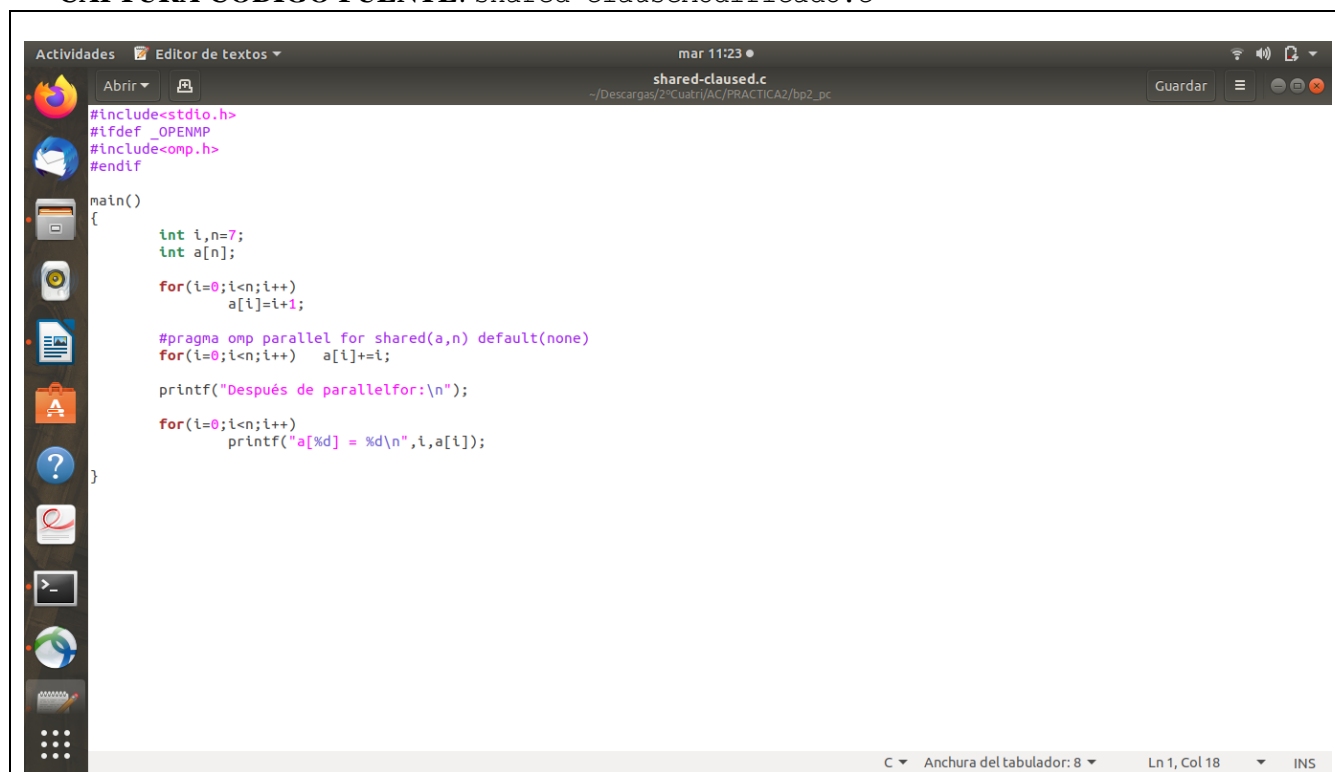
Antes de comenzar a realizar el trabajo de este cuaderno consultar el fichero con los normas de prácticas que se encuentra en SWAD

Ejercicios basados en los ejemplos del seminario práctico

1. ¿Qué ocurre si en el ejemplo del seminario `shared-clause.c` se añade a la directiva `parallel` la cláusula `default(none)`? **(b)** Resuelva el problema generado sin eliminar `default(none)`. Añada el código con la modificación al cuaderno de prácticas. (Añada capturas de pantalla que muestren lo que ocurre)

RESPUESTA: Da un error porque `default(none)` nos obliga a que todas las variables que aparecen en la zona paralelizada tengan especificadas su ámbito. Con `share` se dice que la variable es compartida pero queda decir cuál es el ámbito de `n` que también sería compartida.

CAPTURA CÓDIGO FUENTE: `shared-clauseModificado.c`



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

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

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

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

    printf("Después de parallelfor:\\n");

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

CAPTURAS DE PANTALLA:

```

xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc
JavierRamirezPulido>mar mar 31 gcc -O2 shared-claused.c -o shared_clause
JavierRamirezPulido>mar mar 31 gedit shared-claused.c
^C
JavierRamirezPulido>mar mar 31 gcc -O2 shared-claused.c -o shared_clause
JavierRamirezPulido>mar mar 31 gedit shared-claused.c
^C
JavierRamirezPulido>mar mar 31 gcc -O2 shared-claused.c -o shared_clause
shared-claused.c:6:1: warning: return type defaults to 'int' [-Wimplicit-int]
main()
^
JavierRamirezPulido>mar mar 31 gedit shared-claused.c
^C
JavierRamirezPulido>mar mar 31 gcc -O2 -fopenmp shared-claused.c -o shared_clause
shared-claused.c: In function 'main':
shared-claused.c:14:10: error: 'n' not specified in enclosing 'parallel'
#pragma omp parallel for shared(a) default(none)
^
shared-claused.c:14:10: error: enclosing 'parallel'
JavierRamirezPulido>mar mar 31

```

FALLO EN EJECUCION

```

xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc
JavierRamirezPulido>mar mar 31 gcc -O2 -fopenmp shared-claused.c -o shared_clause
JavierRamirezPulido>mar mar 31 ./shared_clause
Después de parrellefor:
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
a[6] = 13
JavierRamirezPulido>mar mar 31

```

EJECUCION CORRECTA

2. Añadir a lo necesario a `private-clause.c` para que imprima suma fuera de la región `parallel` e inicializar suma a un valor distinto de 0. Ejecute varias veces el código ¿Qué imprime el código fuera del `parallel`? (muéstrelo con una captura de pantalla) ¿Qué ocurre si en esta versión de `private-clause.c` se inicia la variable suma fuera de la construcción `parallel` en lugar de dentro? Razone su respuesta (añada capturas de pantalla que muestren lo que ocurre). Añadir el código con las modificaciones al cuaderno de prácticas.

RESPUESTA: Lo que ocurre es que al inicializar la variable fuera del `parallel`, se le da valor a la variable

que es global/compartida, pero lo que hace `private()` es crear para cada una de las hebras una copia de la variable de tal forma que el valor de esta se queda de manera local en la hebra y no modifica ni actualiza la global. Fuera del `parallel`, lo que queda es el valor que tenía la original, haya pasado lo que haya pasado de por medio.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado.c`

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

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

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

#pragma omp parallel private (suma)
{
    suma=7;
    #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");
}
```

ANTES DE LA MODIFICACION

```

#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=7;
#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*Fuera del parallel thread %d suma= %d", omp_get_thread_num(), suma);
    printf("\n");
}

```

TRAS LA MODIFICACION

CAPTURAS DE PANTALLA:

```

Actividades Terminal mar 11:52
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc
Archivo Editar Ver Buscar Terminal Pestañas Ayuda
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PR... x c3estudiante18@atcgrid:~/bp2 x xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PR...
JavierRamirezPulido>mar mar 31 gcc -O2 -fopenmp private-clause-primer-parte.c -o private-clause
JavierRamirezPulido>mar mar 31 ./private-clause
thread1 suma a[2] / thread1 suma a[3] / thread2 suma a[4] / thread3 suma a[5] / thread0 suma a[6] / thread0 suma a[0] / thread0 suma a[1] /
* thread 1 suma= 12
* thread 3 suma= 13
* thread 0 suma= 8
* thread 2 suma= 16
*Fuera del parallel thread 0 suma= 0
JavierRamirezPulido>mar mar 31 ./private-clause
thread1 suma a[2] / thread1 suma a[3] / thread2 suma a[4] / thread2 suma a[5] / thread3 suma a[6] / thread0 suma a[0] / thread0 suma a[1] /
* thread 1 suma= 12
* thread 2 suma= 16
* thread 3 suma= 13
* thread 0 suma= 8
*Fuera del parallel thread 0 suma= 0
JavierRamirezPulido>mar mar 31 ./private-clause
thread1 suma a[2] / thread1 suma a[3] / thread2 suma a[4] / thread2 suma a[5] / thread3 suma a[6] / thread0 suma a[0] / thread0 suma a[1] /
* thread 0 suma= 8
* thread 1 suma= 12
* thread 2 suma= 16
* thread 3 suma= 13
*Fuera del parallel thread 0 suma= 0
JavierRamirezPulido>mar mar 31

```

PRIMERA PARTE DEL ENUNCIADO

```

Actividades Terminal mar 11:54
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc

JavierRamirezPulido>mar mar 31 ./private-clausemodificado
thread3 suma a[6] / thread1 suma a[2] / thread1 suma a[3] / thread0 suma a[0] / thread0 suma a[1] / thread2 suma a[4] / thread2 suma a[5] /
* thread 1 suma= -360909995
* thread 2 suma= -360909991
* thread 0 suma= -1788521279
* thread 3 suma= -360909994
*Fuera del parallel thread 0 suma= 7
JavierRamirezPulido>mar mar 31 ./private-clausemodificado
thread0 suma a[0] / thread0 suma a[1] / thread1 suma a[2] / thread1 suma a[3] / thread2 suma a[4] / thread2 suma a[5] / thread3 suma a[6] /
* thread 0 suma= 1329795729
* thread 1 suma= -69786795
* thread 2 suma= -69786791
* thread 3 suma= -69786794
*Fuera del parallel thread 0 suma= 7
JavierRamirezPulido>mar mar 31 ./private-clausemodificado
thread1 suma a[2] / thread1 suma a[3] / thread2 suma a[4] / thread2 suma a[5] / thread3 suma a[6] / thread0 suma a[0] / thread0 suma a[1] /
* thread 0 suma= -220432543
* thread 2 suma= -700984487
* thread 1 suma= -700984491
* thread 3 suma= -700984490
*Fuera del parallel thread 0 suma= 7
JavierRamirezPulido>mar mar 31

```

TRAS MODIFICACION

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

RESPUESTA: El resultado es distinto en cada ejecución, pero en cada una de esta, todas las sumas parciales son iguales. Se debe a que el `private` creaba una copia individual por hebra. Ahora, al no haber, cada hebra usa la variable `suma` compartida, inicializada a 0 para cada hebra la modifica. Como no tiene una cláusula de sincronización, se produce condición de carrera y por ello los resultados son impredecibles.

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=7;
#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:

```

xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc
JavierRamirezPulido>mar mar 31 gcc -O2 -fopenmp private-clauseModificado3.c -o private-clausemodificado3
JavierRamirezPulido>mar mar 31 ./private-clausemodificado3
thread1 suma a[2] / thread1 suma a[3] / thread2 suma a[4] / thread0 suma a[0] / thread0 suma a[1] / thread2 suma a[5] / thread3 suma a[6] /
* thread 2 suma= 13
* thread 3 suma= 13
* thread 1 suma= 13
* thread 0 suma= 13
JavierRamirezPulido>mar mar 31 ./private-clausemodificado3
thread1 suma a[2] / thread1 suma a[3] / thread3 suma a[6] / thread2 suma a[4] / thread2 suma a[5] / thread0 suma a[0] / thread0 suma a[1] /
* thread 0 suma= 17
* thread 1 suma= 17
* thread 3 suma= 17
* thread 2 suma= 17
JavierRamirezPulido>mar mar 31 ./private-clausemodificado3
thread2 suma a[4] / thread2 suma a[5] / thread0 suma a[0] / thread0 suma a[1] / thread1 suma a[2] / thread1 suma a[3] / thread3 suma a[6] /
* thread 2 suma= 11
* thread 0 suma= 11
* thread 1 suma= 11
* thread 3 suma= 11
JavierRamirezPulido>mar mar 31

```

4. En la ejecución de `firstlastprivate.c` de la pag. 21 del seminario se imprime un 6 fuera de la región `parallel`. ¿El código imprime siempre 6 fuera de la región `parallel`? Razone su respuesta (añada capturas de pantalla que muestren lo que ocurre).

RESPUESTA: Siempre imprime 6 si se ejecuta con un número de hebras concreto todas las veces. A la que cambies el número de hebras, y por tanto el numero de iteraciones que hace cada una, se modifica el valor. Es el factor del que depende y en las capturas a continuación se demuestra.

CAPTURAS DE PANTALLA:

```

xavivi2000@xavivi2000: ~/Descargas/2°Cuatri/AC/PRACTICA2/bp2_pc
JavierRamirezPulido>mar mar 31 ./firstlastprivate
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
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
*Fuera de la construccion parallel suma= 6JavierRamirezPulido>mar mar 31 ./firstlastprivate
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
*Fuera de la construccion parallel suma= 6JavierRamirezPulido>mar mar 31 ./firstlastprivate
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 3 suma a[6] suma=6
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
*Fuera de la construccion parallel suma= 6JavierRamirezPulido>mar mar 31

```

EJECUCION EN EL QUE SUMA SIEMPRE = 6

```

xavivi2000@xavivi2000: ~/Descargas/2°Cuatri/AC/PRACTICA2/bp2_pc
JavierRamirezPulido>mar mar 31 export OMP_NUM_THREADS=4
JavierRamirezPulido>mar mar 31 ./firstlastprivate
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 3 suma a[6] suma=6
*Fuera de la construccion parallel suma= 6
JavierRamirezPulido>mar mar 31 export OMP_NUM_THREADS=2
JavierRamirezPulido>mar mar 31 ./firstlastprivate
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 0 suma a[2] suma=3
thread 0 suma a[3] suma=6
thread 1 suma a[4] suma=4
thread 1 suma a[5] suma=9
thread 1 suma a[6] suma=15
*Fuera de la construccion parallel suma= 15
JavierRamirezPulido>mar mar 31 export OMP_NUM_THREADS=3
JavierRamirezPulido>mar mar 31 ./firstlastprivate
thread 1 suma a[3] suma=3
thread 1 suma a[4] suma=7
thread 2 suma a[5] suma=5
thread 2 suma a[6] suma=11
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 0 suma a[2] suma=3
*Fuera de la construccion parallel suma= 11
JavierRamirezPulido>mar mar 31

```

CAMBIANDO EL NUMERO DE HEBRAS PARA VER COMO AFECTA AL RESULTADO

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

RESPUESTA: Cada hebra tendrá una copia privada de la variable y la labor de `single` es que una zona de código concreta se haga de forma secuencial ejecutada por la primera hebra llegada a ese punto. La introducción del valor a la variable se produce dentro del `single`, y la declaración de la variable dentro del `parallel`. A efectos prácticos, esto produce que ahora la variable no es compartida, es privada y cada una tendrá un valor independiente. Esto provoca que solo la hebra que llega primera a `single` consigue introducirle un

valor a la variable. Así, solo una de las copias tiene un valor válido al salir del single. Solo estarán bien los resultados obtenidos por la hebra que llego primero (en nuestro caso es la hebra que realiza la iteración 3, 4 y 5)

CAPTURA CÓDIGO FUENTE: copyprivate-clauseModificado.c

```

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

main() {
    int n=9, i, b[n];
    for(i=0; i<n; i++) b[i]=-1;

    #pragma omp parallel
    {
        int a;
        #pragma omp single
        {
            printf("\nIntroduce valor de inicialización a: ");
            scanf("%d", &a);
            printf("\nSingle ejecutada por el thread%d\n", omp_get_thread_num());
        }

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

    printf("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:

```

xavivi2000@xavivi2000: ~/Descargas/2°Cuatri/AC/PRACTICA2/bp2_pc
JavierRamirezPulido> mar 31 ./copyprivate-clause
Introduce valor de inicialización a: 15
Single ejecutada por el thread1
Después de la región parallel:
b[0] = 0    b[1] = 0    b[2] = 0    b[3] = 15    b[4] = 15    b[5] = 15    b[6] = 0    b[7] = 0    b[8] = 0
JavierRamirezPulido> mar 31 ./copyprivate-clause
Introduce valor de inicialización a: 12
Single ejecutada por el thread1
Después de la región parallel:
b[0] = 0    b[1] = 0    b[2] = 0    b[3] = 12    b[4] = 12    b[5] = 12    b[6] = 0    b[7] = 0    b[8] = 0
JavierRamirezPulido> mar 31

```

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: Se imprime el mismo resultado que se imprime cuando ejecutamos con suma=0 pero sumando 10 a cada resultado anterior. Igualmente, si cambiásemos ese 10 por 500 y probamos, el resultado será el mismo que el original pero 500 unidades mayor. Se debe a que reduction permite eficientemente calcular el

mismo resultado que ir sumando en una sección crítica las sumas parciales. Se declara suma compartida al ponerla por encima de parallel y reduction lo que hace es una agrupación con la operación de suma y la variable llamada suma. De esta manera, cada hebra tendrá su copia privada y conforme van terminando de ejecutarse las hebras se obtiene un punto de sincronización y se produce el agrupamiento. El mismo resultado que atomic pero más eficiente. Resulta que con reduction haces sumas parciales MAS LA SUMA DEL VALOR INICIAL, por eso si empezaba en 0, el resultado era el resultado de las sumas, pero al empezar en 10 hace las mismas sumas, pero al añadir el valor inicial aumenta 10.

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

main(int argc,char**argv) {

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

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

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

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

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

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

}
```

CAPTURAS DE PANTALLA:

The screenshot shows a terminal window with the following content:

```
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc
Archivo Editar Ver Buscar Terminal Pestañas Ayuda
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PR... x c3estudiante18@atcgrid: ~/bp2 x xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PR... x
JavierRamirezPulido>mar mar 31 ./reduction-clause_con_0 6
Tras 'parallel' suma=15
JavierRamirezPulido>mar mar 31 ./reduction-clause_con_0 20
Tras 'parallel' suma=190
JavierRamirezPulido>mar mar 31 ./reduction-clause_con_0 10
Tras 'parallel' suma=45
JavierRamirezPulido>mar mar 31 ./reduction-clause_con_10 6
Tras 'parallel' suma=25
JavierRamirezPulido>mar mar 31 ./reduction-clause_con_10 20
Tras 'parallel' suma=200
JavierRamirezPulido>mar mar 31 ./reduction-clause_con_10 10
Tras 'parallel' suma=55
JavierRamirezPulido>mar mar 31
```

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: Una forma de ver lo eficiente y compacta que es la clausula `reduction` es observar las modificaciones necesarias para obtener un mismo resultado. El caso sería crear una variable que contendría la suma local de cada hebra y una clausula de sincronización llamada `atomic` para que no se produzca situación de carrera y que la suma sea en exclusión mutua.

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

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:

```

xavivi2000@xavivi2000: ~/Descargas/2°Cuatri/AC/PRACTICA2/bp2_pc
JavierRamirezPulido>mar mar 31 ./reduction-clause 16
Tras 'parallel' suma=120
JavierRamirezPulido>mar mar 31 ./reduction-clause 10
Tras 'parallel' suma=45
JavierRamirezPulido>mar mar 31

```

Resto de ejercicios

8. Implementar un programa secuencial en C que calcule el producto de una matriz cuadrada, M , por un vector, $v1$ (implemente una versión para variables globales y otra para variables dinámicas, use una de estas versiones en los siguientes ejercicios):

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

NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada al programa; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, $v3$, para tamaños pequeños de los vectores (por ejemplo, $N = 8$ y $N=11$); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código paralelo que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

CAPTURA CÓDIGO FUENTE: `pmv-secuencial.c`

```

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

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

    double t_ini, t_fin, resultado_final, final;
    int i,j;

    if(argc<=1){

        printf("Tienen que ser dos argumentos\n");
        exit(-1);

    }

    unsigned int N= atoi(argv[1]);

    double *primer_vector, *segundo_vector, **M;

    segundo_vector=(double*) malloc(N*sizeof(double));
    primer_vector=(double*) malloc(N*sizeof(double));
    M = (double**) malloc(N*sizeof(double *));

    if((M==NULL) || (segundo_vector==NULL) || (primer_vector==NULL) ){

        printf("Fallo reservando los vectores\n");
        exit(-2);

    }

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

        M[i] = (double*) malloc(N*sizeof(double));

        if (M[i]==NULL){

            printf("Fallo reservando los vectores\n");
            exit(-2);

        }

    }

}

```

```

//Inicializar matriz y vectores

for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        M[i][j]=1;
    }
}

for(i=0; i<N; i++){
    primer_vector[i]=1;
}

t_ini = omp_get_wtime();

for(i=0; i<N; i++){
    final=0;
    for(j=0; j<N; j++){
        final+=(primer_vector[j] * M[i][j]);
    }
    segundo_vector[i]=final;
}

t_fin = omp_get_wtime();

t_fin = omp_get_wtime();

resultado_final = t_fin-t_ini;

printf("Tiempo(seg.): %11.9f   Tamano:%u   segundo_vector[0]=%8.6f   segundo_vector[%d]=%8.6f\n", resultado_final,N,segundo_vector[0],N-1,segundo_vector[N-1]);

if (N<20){
    for(i=0; i<N; i++)
        printf("%f ", segundo_vector[i]);
}

printf("\n");

//Liberar memoria

free(segundo_vector);

free(primer_vector);

//Liberar memoria de la matriz

for(j=0; j<N; j++) free(M[j]);

free(M);

return 0;
}

```

CAPTURAS DE PANTALLA:

```

Actividades Terminal mar 13:59
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc/ejer8
JavierRamirezPulido>mar mar 31 ./ejer8 8
Tiempo(seg.): 0.000000818 / Tamaño:8 / segundo_vector[0]=8.000000 segundo_vector[7]=8.000000
8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000
JavierRamirezPulido>mar mar 31 ./ejer8 14
Tiempo(seg.): 0.000001425 / Tamaño:14 / segundo_vector[0]=14.000000 segundo_vector[13]=14.000000
14.000000 14.000000 14.000000 14.000000 14.000000 14.000000 14.000000 14.000000 14.000000 14.000000 14.000000 14.000000 14.000000 14.000000
JavierRamirezPulido>mar mar 31

```

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).

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

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4
5  int main(int argc, char** argv){
6
7      if(argc<=1){
8          .....
9          printf("Mete dos argumentos\n");
10         exit(-1);
11     }
12
13     double t_ini;
14     double t_fin;
15     double resultado_final;
16     double final;
17     int i,j;
18     unsigned int Dim= atoi(argv[1]);
19
20     double *primer_vector=(double*) malloc(Dim*sizeof(double));
21     double *segundo_vector=(double*) malloc(Dim*sizeof(double));
22     double **M(double**) malloc(Dim*sizeof(double *));
23
24
25     if( (M==NULL) || (segundo_vector==NULL) || (primer_vector==NULL)){
26         .....
27         printf("Error en la reserva\n");
28         exit(-2);
29     }
30
31     for (i=0; i<Dim; i++){
32         .....
33         M[i] = (double*) malloc(Dim*sizeof(double));
34
35         if (M[i]==NULL){
36             .....
37             printf("Error en la reserva\n");
38             exit(-2);
39         }
40     }
41
42     }
43
44     //Inicializacion
45     ..

```

```
44
45 //Inicializacion
46 #pragma omp parallel
47 {
48     #pragma omp for
49     for(i=0; i<Dim; i++){
50         .....
51         primer_vector[i]=1;
52         .....
53     }
54
55     #pragma omp for private(j)
56     for(i=0; i<Dim; i++){
57         .....
58         for(j=0; j<Dim; j++)
59             M[i][j]=1;
60         .....
61     }
62
63     #pragma omp single
64     {
65         t_ini = omp_get_wtime();
66     }
67
68
69
70
71     for(i=0; i<Dim; i++){
72         .....
73         final=0;
74         segundo_vector[i]=0;
75         .....
76         for(j=0; j<Dim; j++){
77             .....
78             final+=(primer_vector[j]*M[i][j]);
79             .....
80         }
81         .....
82         segundo_vector[i]=final;
83     }
84
```



```

81      segundo_vector[i]=final;
82  }
83
84
85  #pragma omp single
86  {
87      t_fin = omp_get_wtime();
88  }
89
90  }
91
92  resultado_final = t_fin-t_ini;
93
94  printf("Tiempo: %11.9f\n ", resultado_final);
95  printf("Tamano: %u\n", Dim);
96  printf("segundo_vector[0]=%8.6f\n",segundo_vector[0]);
97  printf("segundo_vector[%d]=%8.6f\n", Dim-1,segundo_vector[Dim-1]);
98
99  if (Dim<20){
100
101      for(i=0; i<Dim; i++)
102          printf("%f ", segundo_vector[i]);
103
104
105  }
106
107  printf("\n");
108
109
110  //liberar memoria
111  free(segundo_vector);
112
113  free(primer_vector);
114
115  //liberar memoria de la matriz
116  for(j=0; j<N; j++) free(M[j]);
117
118  free(M);
119
120  return 0;
121
122
123
124
125  }

```

CAPTURA CÓDIGO FUENTE: pmv-OpenMP-b.c

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4
5  int main(int argc, char** argv){
6
7      if(argc<=1){
8
9          printf("Mete dos argumentos\n");
10         exit(-1);
11     }
12
13     double t_ini;
14     double t_fin;
15     double resultado_final;
16     double final;
17     int i,j;
18     unsigned int Dim= atoi(argv[1]);
19
20     double *primer_vector=(double*) malloc(Dim*sizeof(double));
21     double *segundo_vector=(double*) malloc(Dim*sizeof(double));
22     double **M(double**) malloc(Dim*sizeof(double *));
23
24     if( (M==NULL) || (segundo_vector==NULL) || (primer_vector==NULL)){
25
26         printf("Error en la reserva\n");
27         exit(-2);
28     }
29
30     for (i=0; i<Dim; i++){
31
32         M[i] = (double*) malloc(Dim*sizeof(double));
33
34         if (M[i]==NULL){
35
36             printf("Error en la reserva\n");
37             exit(-2);
38         }
39     }
40
41     //Inicializar matriz y vectores

```

```
44
45 //Inicializar matriz y vectores
46 #pragma omp parallel private(i)
47 {
48
49     #pragma omp for
50     for(i=0; i<Dim; i++){
51         primer_vector[i]=1;
52     }
53
54     #pragma omp for
55     for(i=0; i<Dim; i++){
56         for(j=0; j<Dim; j++){
57             M[i][j]=1;
58         }
59     }
60
61     #pragma omp single
62     {
63         t_ini = omp_get_wtime();
64     }
65
66     for(i=0; i<Dim; i++){
67         final=0;
68         segundo_vector[i]=0;
69         #pragma omp for
70         for(j=0; j<Dim; j++){
71             final+=(primer_vector[j]*M[i][j]);
72         }
73         #pragma omp atomic
74         segundo_vector[i]=final;
75     }
76
77
78
79
80
81
82
83
84
```

```

83      }
84
85
86      #pragma omp single
87      {
88          t_fin = omp_get_wtime();
89      }
90
91  }
92
93  resultado_final = t_fin-t_ini;
94
95  printf("Tiempo: %11.9f\n ", resultado_final);
96  printf("Tamano: %u\n", Dim);
97  printf("segundo_vector[0]=%8.6f\n",segundo_vector[0]);
98  printf("segundo_vector[%d]=%8.6f\n", Dim-1,segundo_vector[Dim-1]);
99
100
101  if (Dim<20){
102      for(i=0; i<Dim; i++)
103          printf("%f ", segundo_vector[i]);
104
105  }
106
107  printf("\n");
108
109  //Liberar memoria
110
111  free(segundo_vector);
112
113  free(primer_vector);
114
115  //Liberar memoria de la matriz
116
117  for(j=0; j<N; j++) free(M[j]);
118
119  free(M);
120
121  return 0;
122
123
124
125
126
127 }

```

RESPUESTA: Los errores han sido comunes y básicos. Puntos y coma, corchetes y un mal uso de la inicialización de algunas variables. La solución a ellos ha sido una revisión exhaustiva del código y un par de pruebas para ver si los errores iban desapareciendo conforme trataba de arreglarlos. En ejecución no ha habido ningún error concreto exceptuando la introducción del 0 como un parámetro, recibiendo un core dumped.

CAPTURAS DE PANTALLA:

```

Actividades Terminal mar 18:38
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc/ejer8

Archivo Editar Ver Buscar Terminal Ayuda
JavierRamirezPulido>mar mar 31 g++ -O2 -fopenmp pmv-OpenMP-a.c -o pmv-OpenMP-a
pmv-OpenMP-a.c: In function 'int main(int, char**)':
pmv-OpenMP-a.c:71:27: error: invalid form of '#pragma omp atomic' before ';' token
    segundo_vector[i]=final;
                          ^
pmv-OpenMP-a.c: At global scope:
pmv-OpenMP-a.c:100:2: error: expected unqualified-id before 'return'
    return 0;
    ^~~~~~
pmv-OpenMP-a.c:103:1: error: expected declaration before '}' token
}
^
JavierRamirezPulido>mar mar 31

```

ERRORES DE COMPILACION DEL CODIGO A

```

Actividades Terminal mar 19:22
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc/ejer8

Archivo Editar Ver Buscar Terminal Ayuda
JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-a 10
Tiempo(seg.): 0.00003128 / Tamaño:10 / segundo_vector[0]=10.000000 segundo_vector[9]=10.000000
10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000
JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-a 15
Tiempo(seg.): 0.00003810 / Tamaño:15 / segundo_vector[0]=15.000000 segundo_vector[14]=15.000000
15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000
15.000000 15.000000 15.000000 15.000000 15.000000
JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-a 5
Tiempo(seg.): 0.00001912 / Tamaño:5 / segundo_vector[0]=5.000000 segundo_vector[4]=5.000000
5.000000 5.000000 5.000000 5.000000 5.000000
JavierRamirezPulido>mar mar 31

```

EJECUCION CODIGO A

```

Actividades Terminal mar 18:47
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc/ejer8

JavierRamirezPulido>mar mar 31 g++ -O2 -fopenmp pmv-OpenMP-b.c -o pmv-OpenMP-b
pmv-OpenMP-b.c: In function 'int main(int, char**)':
pmv-OpenMP-b.c:71:27: error: invalid form of '#pragma omp atomic' before ';' token
    segundo_vector[i]=final;
                          ^
pmv-OpenMP-b.c: At global scope:
pmv-OpenMP-b.c:100:2: error: expected unqualified-id before 'return'
    return 0;
    ^~~~~~
pmv-OpenMP-b.c:103:1: error: expected declaration before '}' token
}
^
JavierRamirezPulido>mar mar 31 g++ -O2 -fopenmp pmv-OpenMP-b.c -o pmv-OpenMP-b
pmv-OpenMP-b.c: In function 'int main(int, char**)':
pmv-OpenMP-b.c:71:27: error: invalid form of '#pragma omp atomic' before ';' token
    segundo_vector[i]=final;
                          ^
JavierRamirezPulido>mar mar 31 g++ -O2 -fopenmp pmv-OpenMP-b.c -o pmv-OpenMP-b
pmv-OpenMP-b.c: In function 'int main(int, char**)':
pmv-OpenMP-b.c:71:3: error: expected primary-expression before '{' token
{
^
JavierRamirezPulido>mar mar 31 g++ -O2 -fopenmp pmv-OpenMP-b.c -o pmv-OpenMP-b
JavierRamirezPulido>mar mar 31

```

ERROR COMPILACION B

```

Actividades Terminal mar 19:23
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc/ejer8

JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-b 5
Tiempo(seg.): 0.000014797 / Tamano:5 / segundo_vector[0]=5.000000 segundo_vector[4]=6.000000
5.000000 5.000000 5.000000 5.000000 6.000000
JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-b 10
Tiempo(seg.): 0.000012848 / Tamano:10 / segundo_vector[0]=10.000000 segundo_vector[9]=10.000000
10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000 10.000000
JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-b 15
Tiempo(seg.): 0.000019688 / Tamano:15 / segundo_vector[0]=15.000000 segundo_vector[14]=16.000000
15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 16.000000
JavierRamirezPulido>mar mar 31

```

EJECUCION DEL CODIGO B

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

```

1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <omp.h>
4
5  int main(int argc, char** argv){
6
7      if(argc<=1){
8
9          printf("Mete dos argumentos\n");
10         exit(-1);
11     }
12
13     double t_ini;
14     double t_fin;
15     double resultado_final;
16     double final;
17     int i,j;
18     unsigned int Dim= atoi(argv[1]);
19
20     double *primer_vector=(double*) malloc(Dim*sizeof(double));
21     double *segundo_vector=(double*) malloc(Dim*sizeof(double));
22     double **M(double**) malloc(Dim*sizeof(double *));
23
24     if( (M==NULL) || (segundo_vector==NULL) || (primer_vector==NULL)){
25
26         printf("Error en la reserva\n");
27         exit(-2);
28     }
29
30     for (i=0; i<Dim; i++){
31
32         M[i] = (double*) malloc(Dim*sizeof(double));
33
34         if (M[i]==NULL){
35
36             printf("Error en la reserva\n");
37             exit(-2);
38         }
39     }
40
41     //Inicializacion
42
43
44
45

```

```
45 //Inicializacion
46 #pragma omp parallel private(i)
47 {
48     #pragma omp for
49     for(i=0; i<Dim; i++){
50         primer_vector[i]=1;
51     }
52
53     #pragma omp for private(j)
54     for(i=0; i<Dim; i++){
55         for(j=0; j<Dim; j++)
56             M[i][j]=1;
57     }
58
59     #pragma omp single
60     {
61         t_ini = omp_get_wtime();
62     }
63
64     for(i=0; i<Dim; i++){
65         #pragma omp for reduction(+:final)
66         for(j=0; j<Dim; j++){
67             final+=(primer_vector[j]*M[i][j]);
68         }
69         #pragma omp single
70         {
71             segundo_vector[i]=final;
72             final=0;
73         }
74     }
75
76     #pragma omp single
77     {
78         t_fin = omp_get_wtime();
79     }
80 }
```



```

87         t_fin = omp_get_wtime();
88     }
89
90 }
91
92 resultado_final = t_fin-t_ini;
93
94 printf("Tiempo: %11.9f\n ", resultado_final);
95 printf("Tamano: %u\n", Dim);
96 printf("segundo_vector[0]=%8.6f\n",segundo_vector[0]);
97 printf("segundo_vector[%d]=%8.6f\n", Dim-1,segundo_vector[Dim-1]);
98
99
100 if (Dim<20){
101     for(i=0; i<Dim; i++)
102         printf("%f ", segundo_vector[i]);
103
104 }
105
106 printf("\n");
107
108 //Liberar memoria
109 free(segundo_vector);
110
111 free(primer_vector);
112
113 //Liberar memoria de la matriz
114 for(j=0; j<N; j++) free(M[j]);
115
116 free(M);
117
118 return 0;
119
120 }
121
122 }
123
124 }
125
126 }
127
128 }
129
130 }

```

RESPUESTA: Los errores de compilación en este caso estaban ya resueltos por partir del código correcto del apartado b del ejercicio 9. Sin embargo, en la ejecución, el valor 0 volvió a producir una violación del segmento.

CAPTURAS DE PANTALLA:

```

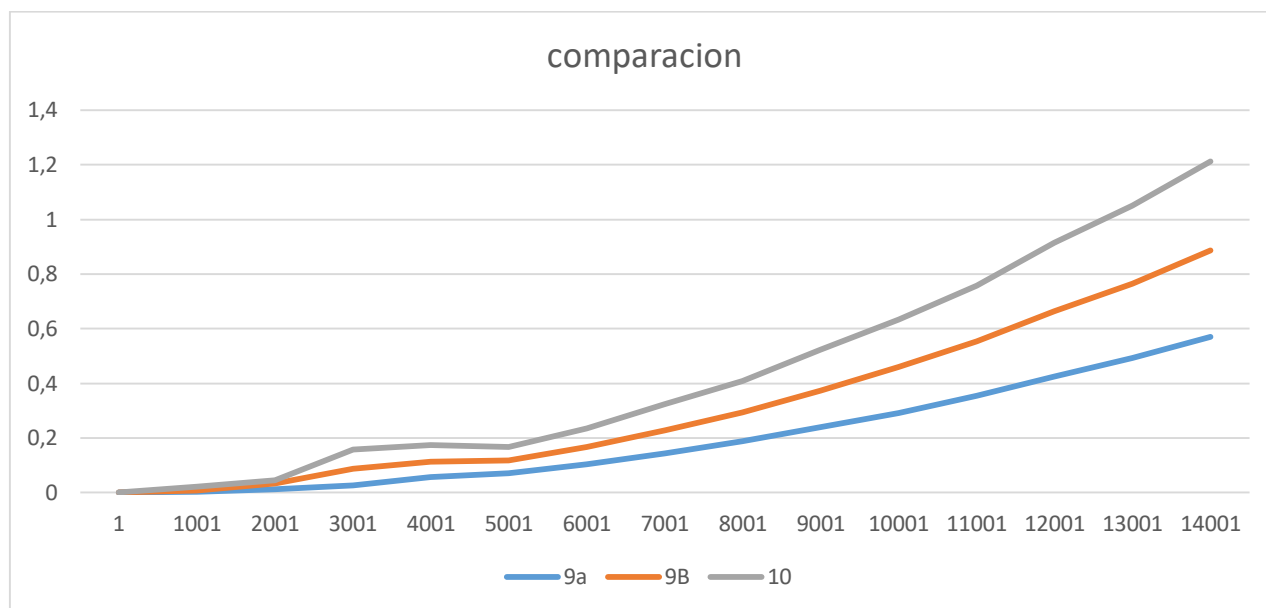
xavivi2000@xavivi2000: ~/Descargas/2ºCuatri/AC/PRACTICA2/bp2_pc/ejer8
JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-reduction 8
Tiempo(seg.): 0.000018045 / Tamaño:8 / segundo_vector[0]=8.000000 segundo_vector[7]=8.000000
8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000
JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-reduction 11
Tiempo(seg.): 0.000024165 / Tamaño:11 / segundo_vector[0]=11.000000 segundo_vector[10]=11.000000
11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000
JavierRamirezPulido>mar mar 31 ./pmv-OpenMP-reduction 15
Tiempo(seg.): 0.000032558 / Tamaño:15 / segundo_vector[0]=15.000000 segundo_vector[14]=15.000000
15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000 15.000000
JavierRamirezPulido>mar mar 31

```

11. Ayudándose de una hoja de cálculo (recuerde que en las aulas está instalado OpenOffice) realice una tabla y una gráfica que permitan comparar la escalabilidad (ganancia en velocidad en función del número de cores) en atcgrid y en 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):

tamaño	9A	9B	10
1	0,000001612	0,00000363	0,00000807
1001	0,002520395	0,005245269	0,012451557
2001	0,012395389	0,020043227	0,012381973
3001	0,025467685	0,061948727	0,069045015
4001	0,055697945	0,057667653	0,060145786
5001	0,071265857	0,045333305	0,049611208
6001	0,104366536	0,062394326	0,067941193
7001	0,143308083	0,08366913	0,096762639
8001	0,186863893	0,106996741	0,113407723
9001	0,238802473	0,134597415	0,149025552
10001	0,290628051	0,169397255	0,172475743
11001	0,354641037	0,197923851	0,204094745
12001	0,425294698	0,238893275	0,251013938
13001	0,492979843	0,27219183	0,284549645
14001	0,569954523	0,316378237	0,326958525

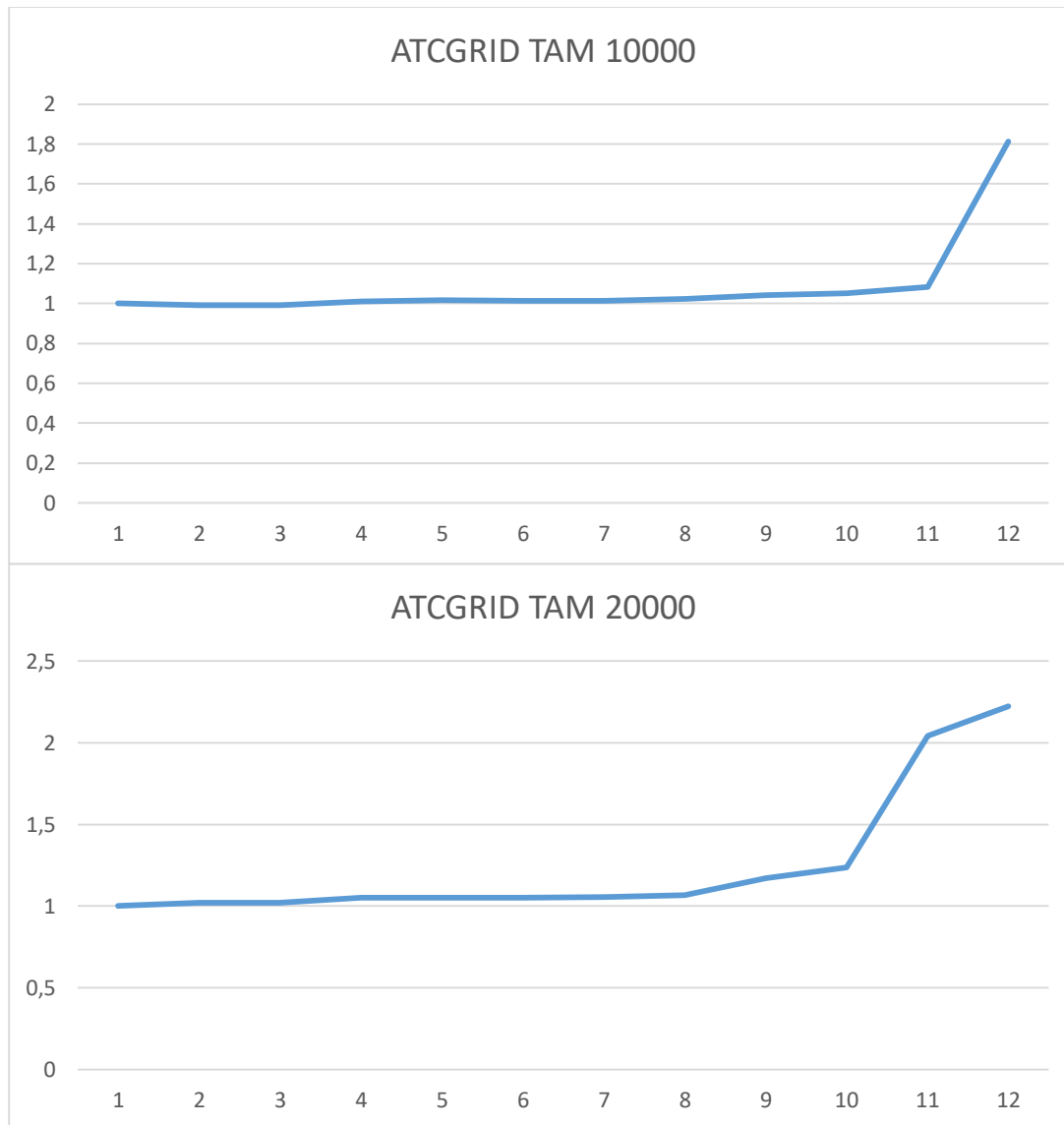


Tras observar y estudiar la tabla y gráfica que compara los tiempos que tardan los 3 códigos para unos tamaños de entrada similares, llego a la conclusión de que el más eficiente es el código del 9.a.

TABLA (con tiempos y ganancia) Y GRÁFICA (con ganancia) (para 1-4 threads PC local, y para 1-12 threads en atcgrid, tamaños-N:- un N entre 20000 y 100000, y otro entre 5000 y 20000):

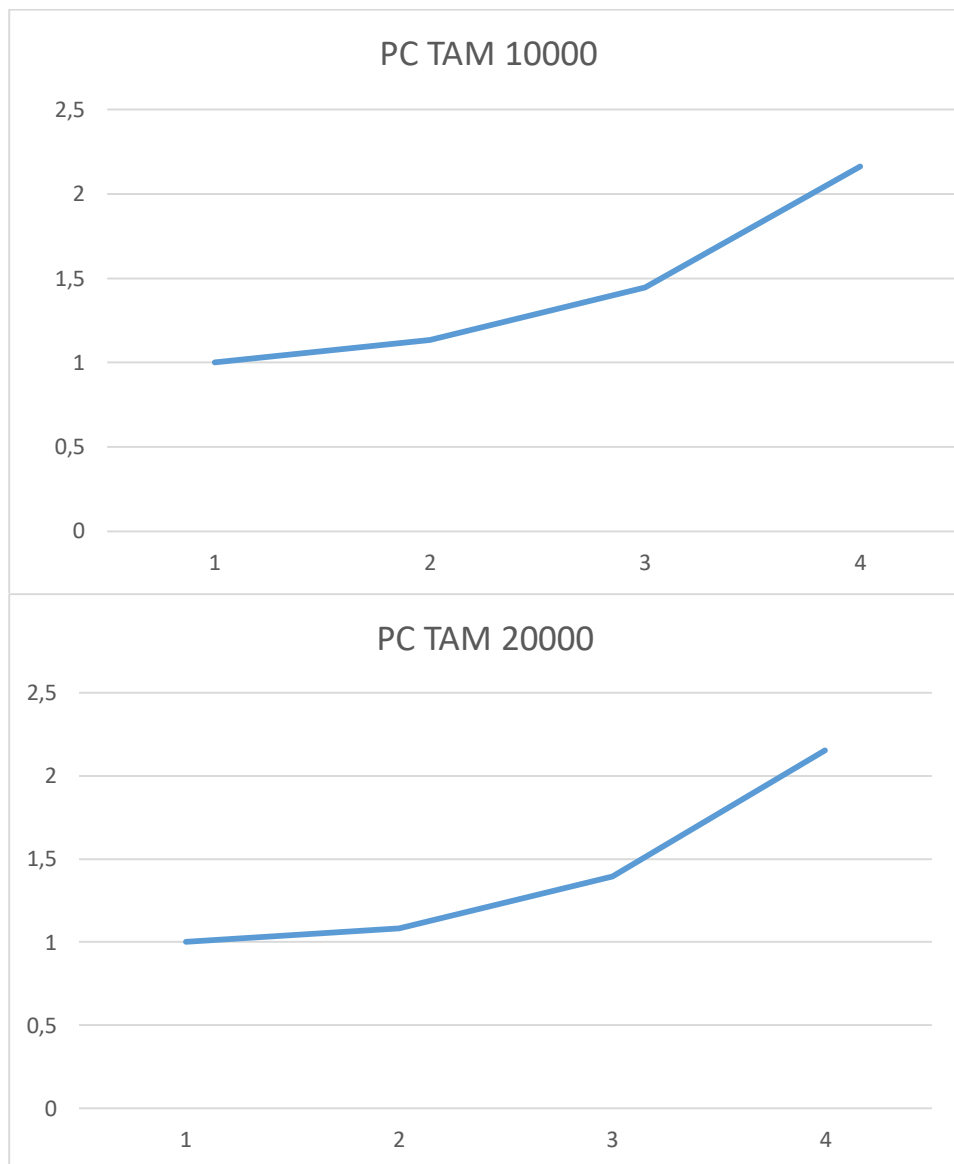
ATCGRID:

Nº HEBRAS	TAM=10000	GANANCIA	TAM=20000	GANANCIA
1	0,110699831	1	0,428389382	1
2	0,109638171	0,990409561	0,437949657	1,022316788
3	0,109620726	0,990251972	0,438040866	1,022529699
4	0,111969918	1,011473251	0,450189466	1,050888479
5	0,112431219	1,015640385	0,450892298	1,052529117
6	0,112199472	1,013546913	0,451364631	1,053631696
7	0,112176562	1,013339957	0,451672228	1,054349727
8	0,113441136	1,024763407	0,457958106	1,069023009
9	0,115221897	1,0408498	0,502370523	1,172696019
10	0,116565742	1,05298934	0,529826313	1,236786754
11	0,119950429	1,083564698	0,87465057	2,041718602
12	0,200613193	1,812226732	0,952041762	2,222374788



PC:

Nº HEBRAS	PC TAM=10000	GANANCIA	PC TAM=20000	GANANCIA
1	0,25807808	1	1,052579874	1
2	0,292526317	1,133479903	1,138334626	1,081471016
3	0,373312564	1,446510157	1,467041838	1,393758207
4	0,558236074	2,163051097	2,268720989	2,155390812



COMENTARIOS SOBRE LOS RESULTADOS:

Como conclusión, veo que más hebras no significan mayor rendimiento. Mi ordenador es algo antiguo y los tamaños demasiado grandes lo dejaban colgado, por ello no sé qué tan fiables son los datos que se muestran en las gráficas, pero en todas se ve que, a partir de cierto número de hebras, la ganancia no es necesariamente positiva