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): Francisco Domínguez Lorente
Grupo de prácticas y profesor de prácticas: Christian Morillas (B1)
Fecha de entrega: 22/04
Fecha evaluación en clase: 25/04

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: Se produce un error ya que se usa la variable n dentro de la directiva for, pero sin embargo se han anulado todas las reglas generales de las variables dentro de la directiva for. Es por ello que manualmente hay que declarar todas las variables que se usen *(excepto la del bucle for, que por defecto es privada y no tiene en cuenta a default(none)).*

   CAPTURA CÓDIGO FUENTE: `shared-clauseModificado.c`

```
14      #pragma omp parallel for shared(a,n) default(none)
15      for (i=0; i<n; i++)     a[i] += i;
16
17      printf("Después de parallel for:\n");
18
19      for (i=0; i<n; i++)
20         printf("a[%d] = %d\n",i,a[i]);
```

   CAPTURAS DE PANTALLA:

```
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer1] 2019-
04-04 jueves
$gcc -O2 -fopenmp -o shared-clause shared-clause.c
shared-clause.c: In function ‘main’:
shared-clause.c:14:12: error: ‘n’ not specified in enclosing ‘parallel’
    #pragma omp parallel for shared(a) default(none)
            ^~~
shared-clause.c:14:12: error: enclosing ‘parallel’
```

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: Ocurre que se mostrará el valor al que estaba inicializado la variable *suma*. En mi caso, he inicializado *suma=2*, por lo que al imprimir la suma después de la directiva *parallel*, el valor de *suma* sigue siendo 2 en vez de 0, como ocurriría si declaramos la variable dentro de la construcción *parallel*.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado.c`

```
11    int a[n], suma=2;
12
13    for (i=0; i<n; i++)
14       a[i] = i;
15
16  #pragma omp parallel firstprivate(suma)
17  {
18    #pragma omp for
19    for (i=0; i<n; i++)
20    {
21        suma = suma + a[i];
22        printf(
23         "thread %d suma a[%d] / ", omp_get_thread_num(), i);
24    }
25    printf(
26    "\n* thread %d suma= %d", omp_get_thread_num(), suma);
27  }
28
29    printf("\nSuma=%d\n", suma);
```

CAPTURAS DE PANTALLA:

Variable suma declarada fuera de la construcción

```
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer2] 2019-04
-11 jueves
$./private-clause-modificado
thread 0 suma a[0] / thread 0 suma a[1] / thread 3 suma a[6] / thread 1 suma a[2]
/ thread 1 suma a[3] / thread 2 suma a[4] / thread 2 suma a[5] /
* thread 1 suma= 7
* thread 3 suma= 8
* thread 0 suma= 3
* thread 2 suma= 11
Suma=2
```

Variable suma declarada dentro de la construcción

```
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer2] 2019-04
-11 jueves
$./private-clause-modificado
thread 0 suma a[0] / thread 0 suma a[1] / thread 3 suma a[6] / thread 1 suma a[2]
/ thread 1 suma a[3] / thread 2 suma a[4] / thread 2 suma a[5] /
* thread 1 suma= 7
* thread 3 suma= 8
* thread 0 suma= 3
* thread 2 suma= 11
Suma=32640
```

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

   RESPUESTA: Ocurre que el valor de todas las sumas parciales es el mismo. Esto se debe a que todas las hebras comparten ahora la variable *suma*, y por tanto al imprimir el valor de la variable, se imprime el último valor que esta ha tomado.

   CAPTURA CÓDIGO FUENTE: `private-clauseModificado3.c`

```
16  #pragma omp parallel
17  {
18    suma=0;
19    #pragma omp for
20    for (i=0; i<n; i++)
21    {
22        suma = suma + a[i];
23        printf(
24         "thread %d suma a[%d] / ", omp_get_thread_num(), i);
25    }
26    printf(
27    "\n* thread %d suma= %d", omp_get_thread_num(), suma);
28  }
```

CAPTURAS DE PANTALLA:

```
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer3] 2019-04
-11 jueves
$./private-clause-modificado3
thread 3 suma a[6] / thread 0 suma a[0] / thread 0 suma a[1] / thread 1 suma a[2]
/ thread 1 suma a[3] / thread 2 suma a[4] / thread 2 suma a[5] /
* thread 3 suma= 15
* thread 0 suma= 15
* thread 2 suma= 15
* thread 1 suma= 15
```

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: No, depende del reparto que se haga, número de hebras… Si la hebra de la última iteración ejecuta más de una iteración, el valor será distinto a 6.

CAPTURAS DE PANTALLA:
Con *n=8;*

```
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer4] 2019-04
-11 jueves
$./firstlastprivate-clause
 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 3 suma a[7] suma=13
 thread 2 suma a[4] suma=4
 thread 2 suma a[5] suma=9

Fuera de la construcción parallel suma=13
```

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: Lo que hace la cláusula *copyprivate(a)* es difundir el valor de *a* a todas las hebras. Si eliminamos la cláusula, esta difusión no se produce y por tanto solo se inicializan correctamente aquellas componentes del vector a las que haya accedido la hebra que ha ejecutado la construcción *single*.

CAPTURA CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

```
 9  #pragma omp parallel
10  {   int a;
11      #pragma omp single
12      {
13          printf("\nIntroduce valor de inicialización a: ");
14          scanf("%d", &a );
15          printf("\nSingle ejecutada por el thread %d\n",
16                  omp_get_thread_num());
17      }
18      #pragma omp for
19      for (i=0; i<n; i++)  b[i] = a;
20  }
21
22     printf("Depués de la región parallel:\n");
23     for (i=0; i<n; i++) printf("b[%d] = %d\t",i,b[i]);
24     printf("\n");
```

CAPTURAS DE PANTALLA:

```
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer5] 2019-04
-11 jueves
$./copyprivate-clause

Introduce valor de inicialización a: 3

Single ejecutada por el thread 1
Depués de la región parallel:
b[0] = 0        b[1] = 0        b[2] = 0        b[3] = 3        b[4] = 3        b[
5] = 0  b[6] = 0        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: Muestra el mismo resultado que la ejecución con *suma=0*, pero sumándole 10 a este resultado. Esto se debe a que la cláusula *reduction* mantiene el valor inicial de la variable.

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado.c`

```c
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:

```
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer6] 2019-04
-11 jueves
$./reduction-clause 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:

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado7.c`

```c
int main(int argc, char **argv)  {
   int i, n=20, a[n],suma=0, sumatotal=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
   {
    #pragma omp for
      for (i=0; i<n; i++){
        suma += a[i];
      }

      #pragma omp critical
      sumatotal = sumatotal + suma;
   }

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

CAPTURAS DE PANTALLA:

```
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer7] 2019-
04-22 lunes
$gcc -fopenmp -O2 -o reduction-clause reduction-clause-modificado7.c
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer7] 2019-
04-22 lunes
$./reduction-clause 10
Tras 'parallel' suma=180
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer7] 2019-
04-22 lunes
```

Resto de ejercicios

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

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

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

```c
    // Inicializamos la matriz y los vectores
    for(i=0; i<tam; i++){
        v1[i] = 1;
        v2[i] = 0;
    }

    for(i=0; i<tam; i++){
        for(j=0; j<tam; j++){
            m[i][j] = 2;
        }
    }

    // Inicializamos la primera variable de tiempo
    t1 = omp_get_wtime();

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

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

        v2[i] = producto_local;
    }

    // Obtenemos el tiempo total transcurrido
    total = omp_get_wtime() - t1;

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

```c
        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 del vector
    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]);
    }

    free(m);

    return 0;
}
```

CAPTURAS DE PANTALLA:

```
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer8] 2019-
04-22 lunes
$gcc -fopenmp -O2 -o pmv-secuencial pmv-secuencial.c
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer8] 2019-
04-22 lunes
$./pmv-secuencial 5
Tamaño vectores: 5
 Tiempo de ejecución: 0.000001
v2[0] = 10.000000
v2[1] = 10.000000
v2[2] = 10.000000
v2[3] = 10.000000
v2[4] = 10.000000
```

```
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer8] 2019-
04-22 lunes
$./pmv-secuencial 5000
Tamaño vectores: 5000
 Tiempo de ejecución: 0.033213
 Primera componente: 10000.000000
 Última componente: 10000.000000
```

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`

```
30      // Reservamos memoria ahora para las componentes de la matriz
31      #pragma omp parallel for
32      for(i=0; i<tam; i++)
33          m[i] = (double*) malloc(tam*sizeof(double));
34
35  #pragma omp parallel
36  {
37      // Inicializamos la matriz y los vectores
38      #pragma omp for
39      for(i=0; i<tam; i++)
40          v1[i] = 1;
41          v2[i] = 0;
42
43      #pragma omp for private(j) // Para paralelizar el for de dentro
44      for(i=0; i<tam; i++){
45          for(j=0; j<tam; j++){
46              m[i][j] = 2;
47          }
48      }
49
50      // Inicializamos la primera variable de tiempo
51      #pragma omp single
52      {
53          t1 = omp_get_wtime();
54      }
55
56      // Calculamos el producto
57      #pragma omp for private(j) // Para paralelizar el for de dentro
58      for(i=0; i<tam; i++){
59          for(j=0; j<tam; j++){
60              v2[i] = v2[i] + (m[i][j]*v1[j]);
61          }
62      }
63  }
```

```
65      // Obtenemos el tiempo total transcurrido
66      #pragma omp single
67      {
68          total = omp_get_wtime() - t1;
69      }
70
71      // Para tamaños pequeños (hasta tam=11), imprimimos todas las componentes del vector resultante
72      // y el tiempo de ejecución
73      if(tam <= 11){
74          printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n", tam, total);
75
76          for(i=0; i<tam; i++){
77              printf("v2[%i] = %f\n", i, v2[i]);
78          }
79      }
80
81      // Para tamaños superiores, imprimimos el tiempo de ejecución y la primera y última componente del vector
82      else{
83          printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n Primera componente: %f\n Última componente: %f\n", tam, total,
84      }
85
86      // Liberamos memoria
87      free(v1);
88      free(v2);
89
90      for(i=0; i<tam; i++){
91          free(m[i]);
92      }
93
94      free(m);
95
96      return 0;
97  }
```

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

```
30      // Reservamos memoria ahora para las componentes de la matriz
31      #pragma omp parallel for
32      for(i=0; i<tam; i++)
33          m[i] = (double*) malloc(tam*sizeof(double));
34
35  #pragma omp parallel private(i)
36  {
37      // Inicializamos la matriz y los vectores
38      #pragma omp for
39      for(i=0; i<tam; i++)
40          v1[i] = 1;
41          v2[i] = 0;
42
43      #pragma omp for private(j) // Para paralelizar el for de dentro
44      for(i=0; i<tam; i++){
45          for(j=0; j<tam; j++){
46              m[i][j] = 2;
47          }
48      }
49
50      // Inicializamos la primera variable de tiempo
51      #pragma omp single
52      {
53          t1 = omp_get_wtime();
54      }
55
56      // Calculamos el producto
57      for(i=0; i<tam; i++){
58          double producto_local = 0;
59
60          #pragma omp for
61          for(j=0; j<tam; j++){
62              producto_local = producto_local + (m[i][j]*v1[j]);
63          }
```

```
65          #pragma omp critical
66          v2[i] += producto_local;
67      }
68  }
69
70      // Obtenemos el tiempo total transcurrido
71      #pragma omp single
72      {
73          total = omp_get_wtime() - t1;
74      }
75
76      // Para tamaños pequeños (hasta tam=11), imprimimos todas las componentes del vector resultante
77      // y el tiempo de ejecución
78      if(tam <= 11){
79          printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n", tam, total);
80
81          for(i=0; i<tam; 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 última componente del vector
87      else{
88          printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n Primera componente: %f\n Última componente: %f\n", tam, total
89      }
90
91      // Liberamos memoria
92      free(v1);
93      free(v2);
94
95      for(i=0; i<tam; i++){
96          free(m[i]);
97      }
98
99      free(m);
```

RESPUESTA: He necesitado consultar en StackOverflow la paralelización de los bucles anidados. Una respuesta sugería usar la cláusula *collapse(2)*, pero otra usaba *private()* sobre la variable del bucle de dentro, y he optado por esa al ser más familiar.

CAPTURAS DE PANTALLA:

```
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer9] 2019-
04-22 lunes
$gcc -fopenmp -O2 -o pmv-OpenMP-a pmv-OpenMP-a.c
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer9] 2019-
04-22 lunes
$./pmv-OpenMP-a 5000
Tamaño vectores: 5000
 Tiempo de ejecución: 0.013530
 Primera componente: 10000.000000
 Última componente: 10000.000000
```

```
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer9] 2019-
04-22 lunes
$gcc -fopenmp -O2 -o pmv-OpenMP-b pmv-OpenMP-b.c
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer9] 2019-
04-22 lunes
$./pmv-OpenMP-b 5000
Tamaño vectores: 5000
 Tiempo de ejecución: 0.100393
 Primera componente: 10000.000000
 Última componente: 10000.000000
```

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`

```
35  #pragma omp parallel private(i)
36  {
37      // Inicializamos la matriz y los vectores
38      #pragma omp for
39      for(i=0; i<tam; i++)
40          v1[i] = 1;
41          v2[i] = 0;
42
43      #pragma omp for private(j) // Para paralelizar el for de dentro
44      for(i=0; i<tam; i++){
45          for(j=0; j<tam; j++){
46              m[i][j] = 2;
47          }
48      }
49
50      // Inicializamos la primera variable de tiempo
51      #pragma omp single
52      {
53          t1 = omp_get_wtime();
54      }
55
56      // Calculamos el producto
57      for(i=0; i<tam; i++){
58
59          #pragma omp for reduction(+:producto_local)
60          for(j=0; j<tam; j++){
61              producto_local = producto_local + (m[i][j]*v1[j]);
62          }
63
64          #pragma omp single
65          {
66              v2[i] = producto_local;
67              producto_local = 0;
68          }
69
70      }
```

```
73      // Obtenemos el tiempo total transcurrido
74      #pragma omp single
75      {
76          total = omp_get_wtime() - t1;
77      }
78
79      // Para tamaños pequeños (hasta tam=11), imprimimos todas las componentes del vector resultante
80      // y el tiempo de ejecución
81      if(tam <= 11){
82          printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n", tam, total);
83
84          for(i=0; i<tam; i++){
85              printf("v2[%i] = %f\n", i, v2[i]);
86          }
87      }
88
89      // Para tamaños superiores, imprimimos el tiempo de ejecución y la primera y última componente del vector
90      else{
91          printf("Tamaño vectores: %i\n Tiempo de ejecución: %f\n Primera componente: %f\n Última componente: %f\n", tam, total
92      }
93
94      // Liberamos memoria
95      free(v1);
96      free(v2);
97
98      for(i=0; i<tam; i++){
99          free(m[i]);
100     }
101
102     free(m);
103
104     return 0;
105 }
```

RESPUESTA: Error 1:

```
[FranciscoDomínguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer10] 2019
-04-22 lunes
$gcc -fopenmp -O2 -o pmv-OpenMP-reduction pmv-OpenMP-reduction.c
pmv-OpenMP-reduction.c: In function ‘main’:
pmv-OpenMP-reduction.c:60:17: error: reduction variable ‘producto_local’ is priv
ate in outer context
        #pragma omp for reduction(+:producto_local)
                ^~~
```

Para solventarlo he declarado la variable *producto_local* fuera del *for* para calcular el producto de la matriz.

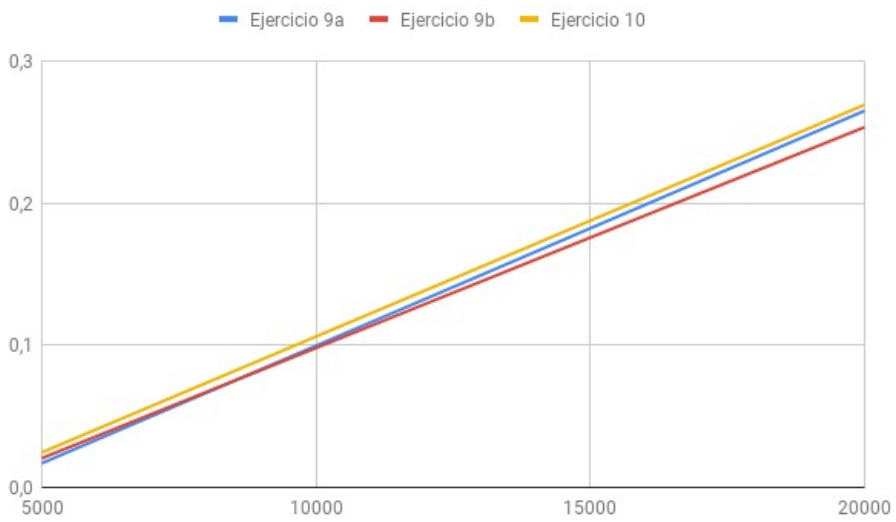
CAPTURAS DE PANTALLA:

```
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer10] 2019
-04-22 lunes
$gcc -fopenmp -O2 -o pmv-OpenMP-reduction pmv-OpenMP-reduction.c
[FranciscoDominguezLorente d3vcho@d3vcho-PC:~/Escritorio/Uni/AC/bp2/ejer10] 2019
-04-22 lunes
$./pmv-OpenMP-reduction 5000
Tamaño vectores: 5000
 Tiempo de ejecución: 0.047862
 Primera componente: 10000.000000
 Última componente: 10000.000000
```

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):

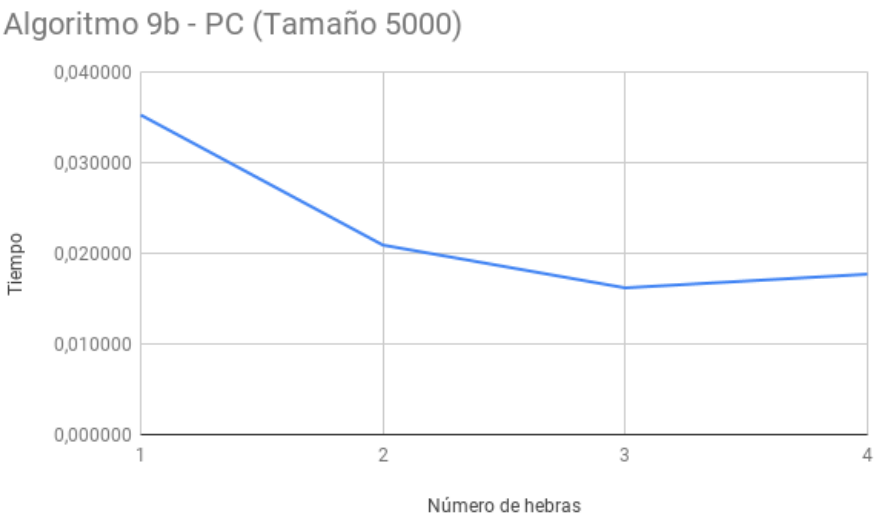
Tamaño	Ejercicio 9a	Ejercicio 9b	Ejercicio 10
5000	0,017006	0,020513	0,024642
20000	0,264906	0,253373	0,269147



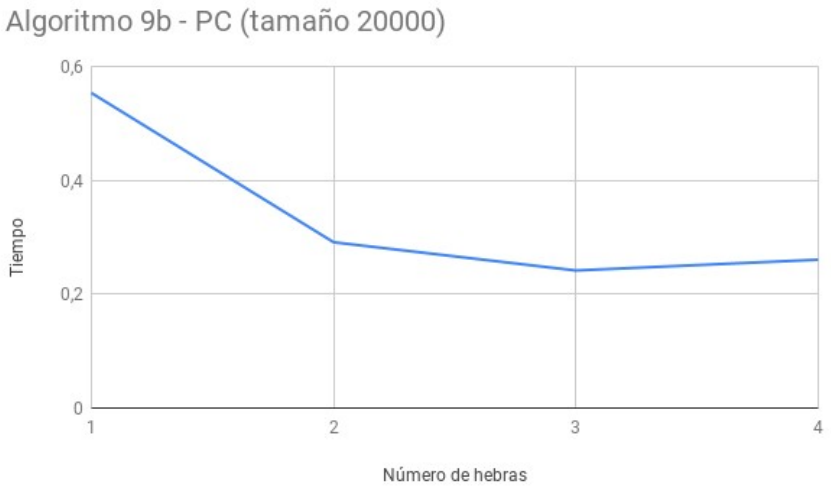
Podemos comprobar que en mi caso, el programa del ejercicio 9b es mejor. Por tanto, lo utilizaré para hacer el estudio.

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):

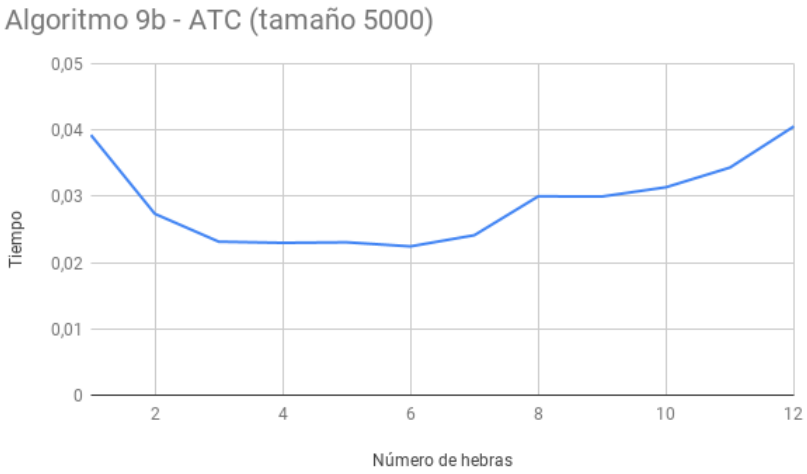
Número de hebras	**Tamaño**	**Tiempo**
1	5000	0,035261
2	5000	0,020906
3	5000	0,016188
4	5000	0,017695



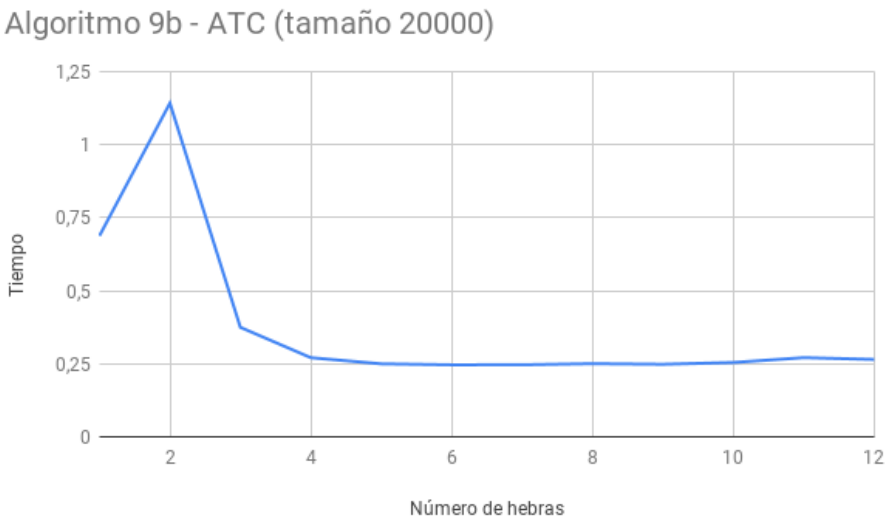
Número de hebras	Tamaño	Tiempo
1	20000	0,553431
2	20000	0,291202
3	20000	0,241610
4	20000	0,260586



Número de hebras	Tamaño	Tiempo
1	5000	0,039245
2	5000	0,027362
3	5000	0,023158
4	5000	0,022984
5	5000	0,023064
6	5000	0,022448
7	5000	0,024132
8	5000	0,029997
9	5000	0,029976
10	5000	0,031370
11	5000	0,034338
12	5000	0,040547



Número de hebras	Tamaño	Tiempo
1	20000	0,688248
2	20000	1,142233
3	20000	0,374966
4	20000	0,270987
5	20000	0,249960
6	20000	0,246598
7	20000	0,246757
8	20000	0,250463
9	20000	0,248429
10	20000	0,254479
11	20000	0,271109
12	20000	0,264956



COMENTARIOS SOBRE LOS RESULTADOS: Se ve fácilmente con las gráficas y con las tablas, que los tiempos de ejecución no siempre mejoran con un mayor número de hebras. Por ejemplo, para tamaño **N=5000**, a partir de 5 hebras, los tiempos de ejecución se vuelven peores que anteriormente.