Grai2º curso / 2º cuatr.

Grado Ing. Inform.

Doble Grado Ing. Inform. y Mat.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos):Antonio Martín Ruiz
Grupo de prácticas:2
Fecha de entrega:20/04/2017
Fecha evaluación en clase:21/04/2017

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)`? (añada una captura de pantalla que muestre lo que ocurre) **(b)** Resuelva el problema generado sin eliminar `default(none)`.Añada el código con la modificación al cuaderno de prácticas.

   RESPUESTA: Se produce un error porque que el ámbito de n no está especificado.

   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) private(n) default(none)
    for(i=0; i<n; i++) a[i] += i;

  printf("Después del parallel for:\n");

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

   CAPTURAS DE PANTALLA:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2$ gcc -O2 -fopenmp share-clause.c -o share-clause
share-clause.c: In function ‘main’:
share-clause.c:14:9: error: ‘n’ not specified in enclosing parallel
 #pragma omp parallel for shared(a) default(none)
         ^
share-clause.c:14:9: error: enclosing parallel
```

2. ¿Qué ocurre si en `private-clause.c` se inicializa la variable `suma` fuera de la construcción `parallel` en lugar de dentro? (inicialice suma a un valor distinto de 0 dentro y fuera de `parallel`) Razone su respuesta. Añada el código con la modificación al cuaderno de prácticas.

   RESPUESTA: La variable suma no estará correctamente inicializada, por lo que contendrá un valor indeterminado que produce errores en el cálculo. Esto se arregla inicializándola dentro de la construcción `parallel`.

   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;//=5;
  for (i=0; i<n; i++)
    a[i] = i;
    #pragma omp parallel private(suma)
    {
      suma=5;
      #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:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2$ gcc -O2 -fopenmp private-clause.c -o private-clause
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2$ ./private-clause
thread 0 suma a[0] / thread 0 suma a[1] / thread 2 suma a[4] / thread 2 suma a[5] / thread 3 suma a[6] / thread 1 suma a[2] / thread 1 suma a[3
] /
* thread 2 suma= 4196569
* thread 3 suma= 4196566
* thread 0 suma= 5
* thread 1 suma= 4196565
```

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2$ gcc -O2 -fopenmp private-clause.c -o private-clause
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2$ ./private-clause
thread 2 suma a[4] / thread 2 suma a[5] / 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= 14
* thread 3 suma= 11
* thread 0 suma= 6
* thread 1 suma= 10
```

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

RESPUESTA: La variable suma es compartida, por lo que cada vez que un hilo la utiliza, lo hace sobreescribiendo el valor anterior, que puede venir de otro hilo. Esto produce errores en los cálculos.

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=5;
      #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:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/Ejercicio3$ gcc -O2 -fopenmp private-clause.c -o private-clause
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/Ejercicio3$ ./private-clause private-clause.c -o private-clause
thread 2 suma a[4] / thread 2 suma a[5] / thread 1 suma a[2] / thread 1 suma a[3] / thread 3 suma a[6] / thread 0 suma a[0] / thread 0 suma a[1
] /
* thread 2 suma= 16
* thread 1 suma= 16
* thread 3 suma= 16
* thread 0 suma= 16
```

4. En la ejecución de `firstlastprivate.c` de la pag. 21 del seminario se imprime un 6 fuera de la región `parallel`. ¿El código imprime siempre 6 fuera de la región `parallel`? Razone su respuesta.

RESPUESTA:
Efectivamente el código siempre imprime 6 fuera de la región parallel. Esto es debido a que firstprivate y lastprivate asignan a la variable `suma` la suma del primer y último elemento del vector, esto es, 0 + 6 = 0.

CAPTURAS DE PANTALLA:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio4$ gcc -O2 -fopenmp firstlastprivate-clause.c -o firstlastprivate
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio4$ ./firstlastprivate
thread 0 suma a[0] /thread 0 suma a[1] /thread 1 suma a[2] /thread 1 suma a[5] /thread 3 suma a[6] /thread 2 suma a[4] /thread 2 suma a[9] /
Fuera de la construccion 'parallel for' suma = 6
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio4$ ./firstlastprivate
thread 0 suma a[0] /thread 0 suma a[1] /thread 3 suma a[6] /thread 1 suma a[2] /thread 1 suma a[5] /thread 2 suma a[4] /thread 2 suma a[9] /
Fuera de la construccion 'parallel for' suma = 6
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio4$ ./firstlastprivate
thread 1 suma a[2] /thread 1 suma a[5] /thread 0 suma a[0] /thread 0 suma a[1] /thread 2 suma a[4] /thread 2 suma a[9] /thread 3 suma a[6] /
Fuera de la construccion 'parallel for' suma = 6
```

5. ¿Qué ocurre si en `copyprivate-clause.c` se elimina la cláusula `copyprivate(a)` en la directiva `single`? ¿A qué cree que es debido?

RESPUESTA: Lo que ocurre es que la lectura del valor de la variable a se realiza en un único thread, por lo que solo se modifica para ese thread. Para el resto, el valor de la variable es el valor que tuviera antes de la lectura, en este caso indeterminado, por lo que se produce un error en el funcionamiento del programa.

CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

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

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

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

#pragma omp parallel
{ int a;
  #pragma omp single
  {
    printf("\nIntroduce el 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:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio5$ ./copyprivate-clause

Introduce el valor de inicializacion a: 20

Single ejecutada por el thread 0
Despues de la region parallel:
b[0] = 20       b[1] = 20       b[2] = 20       b[3] = 20       b[4] = 20       b[5] = 20       b[6] = 20       b[7] = 20       b[8] = 20

amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio5$ gcc -O2 -fopenmp copyprivate-clause_modificado.c -o copyprivate-clause
_modificado
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio5$ ./copyprivate-clause_modificado

Introduce el valor de inicializacion a: 20

Single ejecutada por el thread 2
Despues de la region parallel:
b[0] = 0        b[1] = 0        b[2] = 0        b[3] = 0        b[4] = 0        b[5] = 20       b[6] = 20       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

RESPUESTA: El programa suma los números de 0 a n-1 donde n es el número de iteraciones que se le dan como argumento. Al sustituir el valor indicado en el enunciado, y debido a la clausula reduction, a la suma se le añade 10, por lo que se imprime la suma que se ha descrito +10.

CÓDIGO FUENTE: `reduction-clauseModificado.c`

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

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

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

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

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

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

}
```

CAPTURAS DE PANTALLA:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio6$ gcc -O2 -fopenmp reduction-clause.c -o reduction-clause
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio6$ ./reduction-clause
Falta iteraciones
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio6$ ./reduction-clause 10
Tras 'parallel' suma = 55
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio6$ ./reduction-clause 0
Tras 'parallel' suma = 10
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio6$ ./reduction-clause 1
Tras 'parallel' suma = 10
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio6$ ./reduction-clause 2
Tras 'parallel' suma = 11
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio6$ ./reduction-clause 3
Tras 'parallel' suma = 13
```

7. En el ejemplo `reduction-clause.c`, elimine `reduction()` de `#pragma omp parallel for reduction(+:suma)` y haga las modificaciones necesarias para que se siga realizando la suma de los componentes del vector a en paralelo sin usar directivas de trabajo compartido .

RESPUESTA: Debido a que se elimina `reduction(+:suma)` tenemos que evitar que todas las hebras accedan a suma y modifiquen su valor. Por lo tanto creamos una variable `sumalocal` que almacena la suma que realiza cada hebra y luego utilizando `atomic` evitamos que varias hebras accedan a sumar su parte a la suma total a la vez. De esta manera la suma se realiza correctamente,

CÓDIGO FUENTE: `reduction-clauseModificado7.c`

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

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

  if(argc < 2){
    fprintf(stderr, "Falta iteraciones\n");
```

```
    exit(-1);
  }

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

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

  #pragma omp parallel private (sumalocal)
  {
    sumalocal = 0;
    #pragma omp for
      for (int i = 0; i < n; i++) sumalocal += a[i];

  #pragma omp atomic
    suma += sumalocal;
  }

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

}
```

CAPTURAS DE PANTALLA:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio7$ gcc -O2 -fopenmp reduction-clauseModificado7.c -o reduction-clauseModificado7
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio7$ ./reduction-clauseModificado7
Falta iteraciones
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio7$ ./reduction-clauseModificado7 1
Tras 'parallel' suma = 0
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio7$ ./reduction-clauseModificado7 2
Tras 'parallel' suma = 1
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio7$ ./reduction-clauseModificado7 3
Tras 'parallel' suma = 3
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio7$ ./reduction-clauseModificado7 5
Tras 'parallel' suma = 10
```

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).

CÓDIGO FUENTE: `pmv-secuencial.c`

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

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

```
#else
  #define omp_get_thread_num() 0
  #define omp_get_num_threads() 1
#endif

int main(int argc, char const *argv[]) {

  if (argc < 2){
    printf("ERROR: argumentos incorrectos. Modo de uso: ./pmv-
secuencial <tamaño de matriz y vector>");
    return -1;
  }

    double tiempo_inicio, tiempo_final, tiempo_ejecucion;
    int tamanio = atoi(argv[1]);

    double *vector, *vector_resultado, **matriz;

    vector = (double*) malloc(tamanio*sizeof(double));
    vector_resultado = (double*) malloc(tamanio*sizeof(double));
    matriz = (double**) malloc(tamanio*sizeof(double*));

    if ( (vector == NULL) || (vector_resultado == NULL) || (matriz ==
NULL)){
      printf("Error en la reserva de memoria");
      return -1;
    }

    for (int i = 0; i < tamanio; i++){
      matriz[i] = (double*) malloc(tamanio*sizeof(double));
      if (matriz[i] == NULL){
        printf("Error en la reserva de memoria para matriz");
        return -1;
      }
    }

    for (int i = 0; i < tamanio; i++){
      vector[i] = i;
      vector_resultado[i] = 0;
      for (int j = 0; j < tamanio; j++)
        matriz[i][j] = i+j;
    }

    tiempo_inicio = omp_get_wtime();

    for (int i = 0; i < tamanio; i++){
      for (int j = 0; j < tamanio; j++){
        vector_resultado[i] += matriz[i][j]*vector[j];
      }
    }

    tiempo_final = omp_get_wtime();
    tiempo_ejecucion = tiempo_final - tiempo_inicio;

    printf("Tiempo: %f s. Tamaño: %d. 1 comp: %f. Ult. comp:
%f\n",tiempo_ejecucion, tamanio, vector_resultado[0],
```

```
vector_resultado[tamanio-1]);

    if (tamanio<15){
      for (int i = 0; i < tamanio; i++)
        printf("vector_resultado[%d]=%f", i, vector_resultado[i]);
    }

  return 0;
}
```

CAPTURAS DE PANTALLA:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio8$ gcc -O2 -fopenmp pmv-secuencial.c -o pmv-secuencial
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio8$ ./pmv-secuencial 5
Tiempo: 0.000000 s. Tamaño: 5. 1 comp: 30.000000. Ult. comp: 70.000000
vector_resultado[0]=30.000000vector_resultado[1]=40.000000vector_resultado[2]=50.000000vector_resultado[3]=60.000000vector_resultado[4]=70.0000
00amrantonio@HAL9ecuencial 10
Tiempo: 0.000000 s. Tamaño: 10. 1 comp: 285.000000. Ult. comp: 690.000000
vector_resultado[0]=285.000000vector_resultado[1]=330.000000vector_resultado[2]=375.000000vector_resultado[3]=420.000000vector_resultado[4]=465
.000000vector_resultado[5]=510.000000vector_resultado[6]=555.000000vector_resultado[7]=600.000000vector_resultado[8]=645.000000vector_resultado
[9]=690.000000amrantonio@HAL9000:~ecuencial 20
Tiempo: 0.000001 s. Tamaño: 20. 1 comp: 2470.000000. Ult. comp: 6080.000000
```

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).

CÓDIGO FUENTE : `pmv-OpenMP-a.c`

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

#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
  #define omp_get_num_threads() 1
#endif

int main(int argc, char const *argv[]) {
```

```
  if (argc < 2){
    printf("ERROR: argumentos incorrectos. Modo de uso: ./pmv-
secuencial <tamaño de matriz y vector>");
    return -1;
  }

    double tiempo_inicio, tiempo_final, tiempo_ejecucion;
    int tamanio = atoi(argv[1]);
    int i, j;

    double *vector, *vector_resultado, **matriz;

    vector = (double*) malloc(tamanio*sizeof(double));
    vector_resultado = (double*) malloc(tamanio*sizeof(double));
    matriz = (double**) malloc(tamanio*sizeof(double*));

    if ( (vector == NULL) || (vector_resultado == NULL) || (matriz ==
NULL)){
      printf("Error en la reserva de memoria");
      return -1;
    }

    for ( i = 0; i < tamanio; i++){
      matriz[i] = (double*) malloc(tamanio*sizeof(double));
      if (matriz[i] == NULL){
        printf("Error en la reserva de memoria para matriz");
        return -1;
      }
    }

    #pragma omp parallel
    {

    #pragma omp for private(j)
    for ( i = 0; i < tamanio; i++){
      vector[i] = i;
      vector_resultado[i] = 0;
      for ( j = 0; j < tamanio; j++)
        matriz[i][j] = i+j;
    }

    #pragma omp single
    tiempo_inicio = omp_get_wtime();

    #pragma omp for private(j)
    for ( i = 0; i < tamanio; i++){
      for ( j = 0; j < tamanio; j++){
        vector_resultado[i] += matriz[i][j]*vector[j];
      }
    }

    #pragma omp single
    tiempo_final = omp_get_wtime();

    }
```

```
    tiempo_ejecucion = tiempo_final - tiempo_inicio;


    printf("Tiempo: %f s. Tamaño: %d. 1 comp: %f. Ult. comp:
%f\n",tiempo_ejecucion, tamanio, vector_resultado[0],
vector_resultado[tamanio-1]);

    if (tamanio<15){
      for ( i = 0; i < tamanio; i++)
        printf("vector_resultado[%d]=%f", i, vector_resultado[i]);
    }


  return 0;
}
```

CÓDIGO FUENTE: `pmv-OpenMP-b.c`

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

#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
  #define omp_get_num_threads() 1
#endif

int main(int argc, char const *argv[]) {

  if (argc < 2){
    printf("ERROR: argumentos incorrectos. Modo de uso: ./pmv-
secuencial <tamaño de matriz y vector>");
    return -1;
  }

    double tiempo_inicio, tiempo_final, tiempo_ejecucion;
    int tamanio = atoi(argv[1]);
    int i, j;

    double *vector, *vector_resultado, **matriz;

    vector = (double*) malloc(tamanio*sizeof(double));
    vector_resultado = (double*) malloc(tamanio*sizeof(double));
    matriz = (double**) malloc(tamanio*sizeof(double*));

    if ( (vector == NULL) || (vector_resultado == NULL) || (matriz ==
NULL)){
      printf("Error en la reserva de memoria");
      return -1;
    }

    for ( i = 0; i < tamanio; i++){
      matriz[i] = (double*) malloc(tamanio*sizeof(double));
      if (matriz[i] == NULL){
```

```
        printf("Error en la reserva de memoria para matriz");
        return -1;
      }
    }


    for ( i = 0; i < tamanio; i++){
      vector[i] = i;
      vector_resultado[i] = 0;
      #pragma omp parallel for
      for ( j = 0; j < tamanio; j++)
        matriz[i][j] = i+j;
    }

    tiempo_inicio = omp_get_wtime();



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

      #pragma omp parallel
      {

      double acumulador = 0;
      #pragma omp for
      for ( j = 0; j < tamanio; j++){
        acumulador += matriz[i][j]*vector[j];
      }

      #pragma omp atomic
      vector_resultado[i] += acumulador;
    }

    }

    tiempo_final = omp_get_wtime();



    tiempo_ejecucion = tiempo_final - tiempo_inicio;


    printf("Tiempo: %f s. Tamaño: %d. 1 comp: %f. Ult. comp:
%f\n",tiempo_ejecucion, tamanio, vector_resultado[0],
vector_resultado[tamanio-1]);

    if (tamanio<15){
      for ( i = 0; i < tamanio; i++)
        printf("vector_resultado[%d]=%f", i, vector_resultado[i]);
    }


  return 0;
}
```

RESPUESTA:

Errores:

Durante la compilación de a)

```
pmv-OpenMP-a.c: In function ‘main’:
pmv-OpenMP-a.c:43:29: error: ‘j’ undeclared (first use in this
function)
     #pragma omp for private(j)
```

Solución: declarar j al inicio del programa en lugar de al inicio de los bucles. De esta manera la variable ya está declarada cuando se utiliza la cláusula private.

```
pmv-OpenMP-a.c: In function ‘main’:
pmv-OpenMP-a.c:22:5: error: expected ‘=’, ‘,’, ‘;’, ‘asm’ or
‘__attribute__’ before ‘double’
     double *vector, *vector_resultado, **matriz;
     ^
```

Este error y los posteriores que no se reflejan en la memoria por ser irrelevantes al caso son debidos a la falta de un ; tras la declaración de los enteros i y j que se usan como contadores. La solución es colocar dicho ;.

La ejecución de la versión a ha sido correcta. El resultado es igual al obtenido en la versión sin paralelismo.
Durante la compilación de b)

Sin errores de compilación.

Durante la ejecución de b)

Al ejecutar b se obtiene que tarda mucho en ejecutarse. Tras realizar comprobaciones imprimiendo diferentes mensajes en las diferentes zonas del programa, descubro que es debido a que por equivocación he incluido todo el bucle for que realiza la suma en una sección parallel, lo que hace que cada una de las hebras ejecute el bucle. La solución es simplemente introducir el parallel dentro del bucle for.

Obtenemos unos resultados que no son los correctos. Tras realizar varias comprobaciones imprimiendo las variables que guardan los resultados encuentro que debo usar una variable acumuladora que guarde los resultados que obtiene cada hebra y luego sumarlos todos para cada una de las componentes del vector con una directiva atomic como en el ejercicio 7. De esta manera el resultado es correcto.

CAPTURAS DE PANTALLA:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio9$ gcc -O2 -fopenmp pmv-OpenMP-a.c -o pmv-OpenMP-OpenMP-a
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio9$ ./pmv-OpenMP-OpenMP-a 10
Tiempo: 0.000001 s. Tamaño: 10. 1 comp: 285.000000. Ult. comp: 690.000000
vector_resultado[0]=285.000000vector_resultado[1]=330.000000vector_resultado[2]=375.000000vector_resultado[3]=420.000000vector_resultado[4]=465
.000000vector_resultado[5]=510.000000vector_resultado[6]=555.000000vector_resultado[7]=600.000000vector_resultado[8]=645.000000vector_resultado
[9]=690.000000amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio9$ ./pmv-OpenMP-OpenMP-a 100
Tiempo: 0.000018 s. Tamaño: 100. 1 comp: 328350.000000. Ult. comp: 818400.000000
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio9$ ./pmv-OpenMP-OpenMP-a 1000
Tiempo: 0.001055 s. Tamaño: 1000. 1 comp: 332833500.000000. Ult. comp: 831834000.000000
```

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio9$ ./pmv-OpenMP-OpenMP-b 10
Tiempo: 0.000029 s. Tamaño: 10. 1 comp: 285.000000. Ult. comp: 690.000000
vector_resultado[0]=285.000000vector_resultado[1]=330.000000vector_resultado[2]=375.000000vector_resultado[3]=420.000000vector_resultado[4]=465
.000000vector_resultado[5]=510.000000vector_resultado[6]=555.000000vector_resultado[7]=600.000000vector_resultado[8]=645.000000vector_resultado
[9]=690.000000amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio9$ ./pmv-OpenMP-OpenMP-b 100
Tiempo: 0.000163 s. Tamaño: 100. 1 comp: 328350.000000. Ult. comp: 818400.000000
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio9$ ./pmv-OpenMP-OpenMP-b 1000
Tiempo: 0.002305 s. Tamaño: 1000. 1 comp: 332833500.000000. Ult. comp: 831834000.000000
```

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

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

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

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

#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
  #define omp_get_num_threads() 1
#endif

int main(int argc, char const *argv[]) {

  if (argc < 2){
    printf("ERROR: argumentos incorrectos. Modo de uso: ./pmv-
secuencial <tamaño de matriz y vector>");
    return -1;
  }

    double tiempo_inicio, tiempo_final, tiempo_ejecucion;
    int tamanio = atoi(argv[1]);
    int i, j;

    double *vector, *vector_resultado, **matriz;

    vector = (double*) malloc(tamanio*sizeof(double));
    vector_resultado = (double*) malloc(tamanio*sizeof(double));
    matriz = (double**) malloc(tamanio*sizeof(double*));

    if ( (vector == NULL) || (vector_resultado == NULL) || (matriz ==
NULL)){
      printf("Error en la reserva de memoria");
      return -1;
    }

    for ( i = 0; i < tamanio; i++){
      matriz[i] = (double*) malloc(tamanio*sizeof(double));
      if (matriz[i] == NULL){
        printf("Error en la reserva de memoria para matriz");
        return -1;
      }
    }


    for ( i = 0; i < tamanio; i++){
      vector[i] = i;
```

```
      vector_resultado[i] = 0;
      #pragma omp parallel for
      for ( j = 0; j < tamanio; j++)
        matriz[i][j] = i+j;
    }

    tiempo_inicio = omp_get_wtime();



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

      double acumulador = 0;
      #pragma omp parallel for reduction(+:acumulador)
      for ( j = 0; j < tamanio; j++){
        acumulador += matriz[i][j]*vector[j];
      }

      vector_resultado[i] += acumulador;
    }


    tiempo_final = omp_get_wtime();



    tiempo_ejecucion = tiempo_final - tiempo_inicio;


    printf("Tiempo: %f s. Tamaño: %d. 1 comp: %f. Ult. comp:
%f\n",tiempo_ejecucion, tamanio, vector_resultado[0],
vector_resultado[tamanio-1]);

    if (tamanio<15){
      for ( i = 0; i < tamanio; i++)
        printf("vector_resultado[%d]=%f", i, vector_resultado[i]);
    }


  return 0;
}
```

RESPUESTA: No he tenido errores ni de compilación ni en tiempo de ejecución. Los resultados obtenidos son los esperados. La utilización de la directiva reduction hace más simple el código en este caso.

CAPTURAS DE PANTALLA:

```
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio10$ ./pmv-OpenMP-reduction 10
Tiempo: 0.000013 s. Tamaño: 10. 1 comp: 285.000000. Ult. comp: 690.000000
vector_resultado[0]=285.000000vector_resultado[1]=330.000000vector_resultado[2]=375.000000vector_resultado[3]=420.000000vector_resultado[4]=465
.000000vector_resultado[5]=510.000000vector_resultado[6]=555.000000vector_resultado[7]=600.000000vector_resultado[8]=645.000000vector_resultado
[9]=690.amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio10$ ./pmv-OpenMP-reduction 100
Tiempo: 0.000331 s. Tamaño: 100. 1 comp: 328350.000000. Ult. comp: 818400.000000
amrantonio@HAL9000:~/DGIIM/2DGIIM/AC/Prácticas/Práctica2/src/Ejercicio10$ ./pmv-OpenMP-reduction 1000
Tiempo: 0.001877 s. Tamaño: 1000. 1 comp: 332833500.000000. Ult. comp: 831834000.000000
```

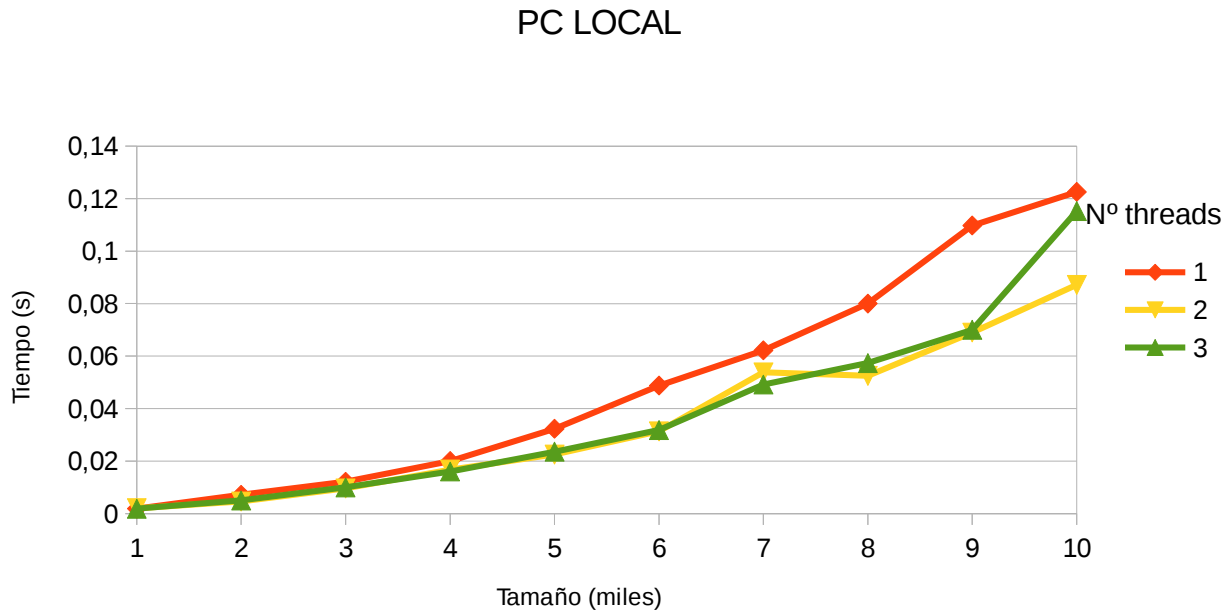
11. 11. Ayudándose de una hoja de cálculo (recuerde que en las aulas está instalado OpenOffice) realice una tabla y una gráfica que permitan comparar la escalabilidad

(ganancia en velocidad en función del número de cores) en atcgrid y en el PC local del mejor código paralelo de los tres implementados en los ejercicios anteriores para dos tamaños (N) distintos (consulte la Lección 6/Tema 2). Usar –O2 al compilar. Justificar por qué el código escogido es el mejor. NOTA: Nunca ejecute en atcgrid código que imprima todos los componentes del resultado.

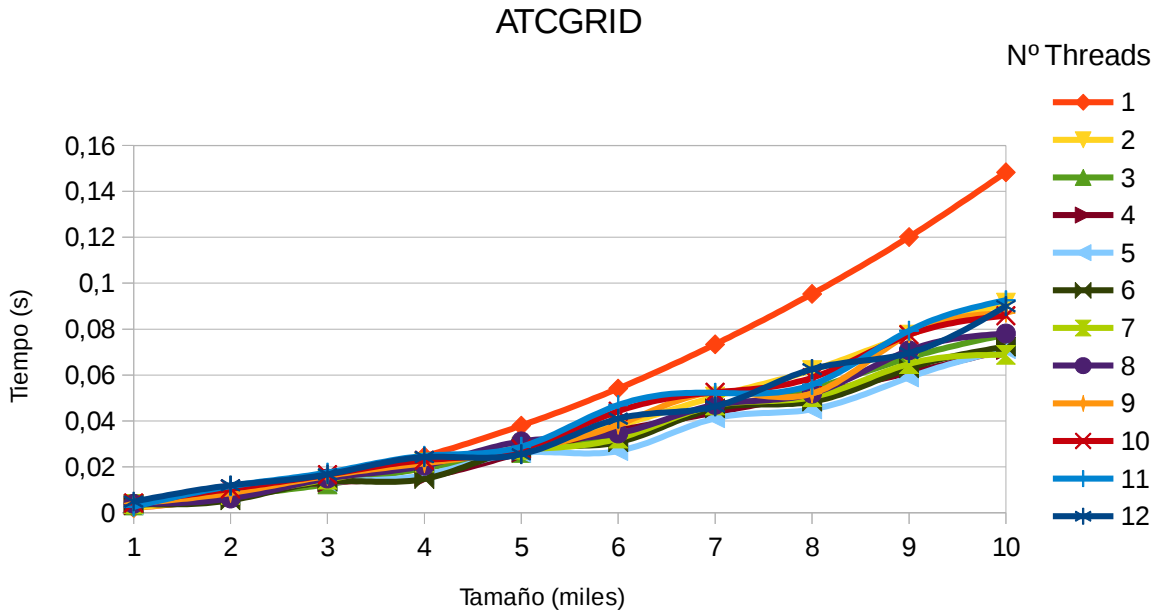
TABLA Y GRÁFICA (por *ejemplo* para 1-4 threads PC local, y para 1-12 threads en atcgrid, tamaños-N-: algúno del orden de cientos de miles):

PC LOCAL

PC LOCAL			
	1	2	3
1000	0,001866	0,002011	0,001868
2000	0,007129	0,004603	0,004996
3000	0,012121	0,00958	0,009957
4000	0,019996	0,016765	0,015997
5000	0,032326	0,022495	0,023566
6000	0,048782	0,031305	0,031817



ATCGRID												
	1	2	3	4	5	6	7	8	9	10	11	12
1000	0,00203	0,002133	0,003096	0,003299	0,003451	0,00381	0,003695	0,003868	0,003987	0,004259	0,002584	0,00485
2000	0,006571	0,006701	0,007442	0,007703	0,007928	0,005423	0,007863	0,006222	0,00861	0,010375	0,011591	0,011904
3000	0,014147	0,0131	0,012364	0,013136	0,013376	0,013473	0,014478	0,014881	0,016056	0,016583	0,017476	0,016696
4000	0,024561	0,019537	0,019441	0,015565	0,016401	0,01469	0,021871	0,020496	0,021554	0,022878	0,024742	0,024229
5000	0,037972	0,028784	0,026106	0,026181	0,026098	0,02788	0,027444	0,031056	0,027407	0,027761	0,028726	0,025462
6000	0,054089	0,038036	0,03454	0,035801	0,026868	0,03072	0,032318	0,034529	0,038316	0,044206	0,046911	0,040971
7000	0,073375	0,050578	0,044299	0,043628	0,041123	0,045463	0,046763	0,046978	0,0524	0,052446	0,052251	0,046333
8000	0,095317	0,062072	0,053933	0,051802	0,045	0,048363	0,050348	0,051886	0,051874	0,058812	0,055743	0,062533
9000	0,120102	0,077531	0,0675	0,060962	0,058941	0,06287	0,064798	0,070757	0,077786	0,077504	0,079301	0,069567
10000	0,148328	0,091489	0,077613	0,070973	0,071325	0,072339	0,068773	0,07805	0,087059	0,085849	0,092586	0,089983



COMENTARIOS SOBRE LOS RESULTADOS:

Tanto para el PC local como para ATCGRID los datos que se piden son demasiado grandes. Por lo tanto he optado por reducir los tamaños de vector y matriz hasta 10000 datos. En las gráficas se aprecia que la escalabilidad es similar para las ejecuciones paralelas. La principal diferencia es entre estas y la ejecución con una sola hebra, que sí obtiene tiempos mucho peores.