

# Arquitectura de Computadores (AC)

## Cuaderno de prácticas.

### Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Álvaro Fernández García

Grupo de prácticas: A2

Fecha de entrega:

Fecha evaluación en clase:

#### 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:** Cuando se introduce la clausula `default` con el parámetro `none`, el programador se ve obligado a especificar cual va a ser el ámbito (compartido o privado) de cada una de las variables, (es decir no se aplica el criterio de la regla general). El array `a` ya está especificado como compartido, la variable `i`, queda excluida de la directiva `default`, ya que se trata del índice de una directiva `for`, sin embargo falta especificar cual va a ser el ámbito de la variable `n`, por tanto el compilador nos da un error.

**CÓDIGO FUENTE:** `shared-clauseModificado.c`

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

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

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

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

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

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

#### CAPTURAS DE PANTALLA:

```
shared-clauseModificado.c: In function 'main':
shared-clauseModificado.c:13:10: error: 'n' not specified in enclosing parallel
    #pragma omp parallel for shared(a), default(none)
    ^
shared-clauseModificado.c:13:10: error: enclosing parallel
```

```
Después de parallel for:
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
a[6] = 13
```

2. ¿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:** Cuando se utiliza la clausula `private`, el valor que toma la variable queda indefinido tanto en la salida como en la entrada aunque esté definida fuera de la construcción. Si inicializamos a 1 la variable `suma` fuera de la construcción `parallel` obtenemos que no se tiene en cuenta ese valor y se opera como si se tratase de un cero (primera captura), sin embargo si la inicializamos a 2 dentro de la construcción, entonces el valor si es tomado en cuenta (segunda captura). En definitiva, si a una variable privada queremos darle un valor inicial debe de hacerse dentro del propio `parallel` y no fuera.

**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 = 1;

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

    #pragma omp parallel private(suma)
    {
        // suma=2;

        #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:**

```
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 a[6] /
* thread 1 suma= 5
* thread 0 suma= 1
* thread 3 suma= 6
* thread 2 suma= 9
```

```
thread 2 suma a[4] / thread 2 suma a[5] / thread 0 suma a[0] / thread 0 suma a[1] / thread 1 suma a[2]
/ thread 1 suma a[3] / thread 3 suma a[6] /
* thread 1 suma= 7
* thread 0 suma= 3
* thread 2 suma= 11
* thread 3 suma= 8
```

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

**RESPUESTA:** Al eliminarlo no se está especificando el ámbito que va a tener la variable `suma`, sin embargo, como esta está declarada fuera de la construcción `parallel` la regla general nos dice que tendrá un ámbito compartido. Tampoco se están realizando accesos controlados por tanto los resultados no son correctos y varían en cada ejecución. Por último todos los threads muestran el mismo resultado por lo mencionado anteriormente, es una variable compartida y en consecuencia todos tienen el mismo valor `suma`.

**CÓDIGO FUENTE:** `private-clauseModificado3.c`

```
#include <stdio.h>

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

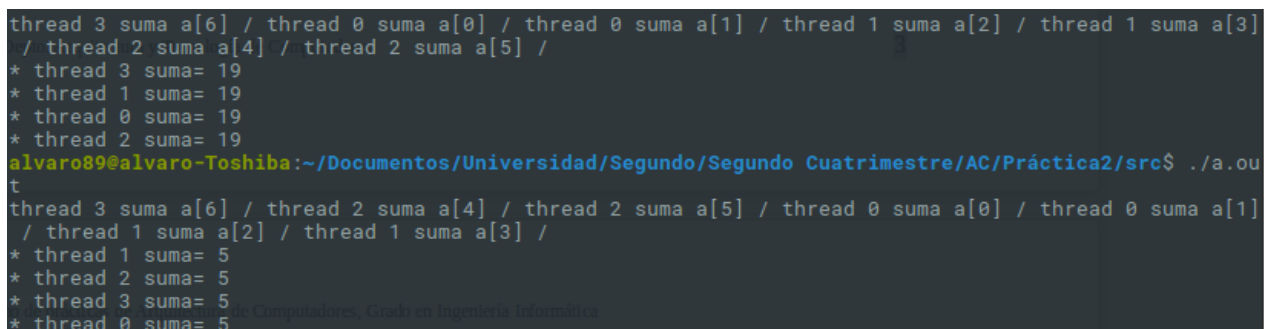
int main(){
    int i, n = 7;
    int a[n], suma;

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

    #pragma omp parallel
    {
        suma=0;

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

**CAPTURAS DE PANTALLA:**



```
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= 19
* thread 1 suma= 19
* thread 0 suma= 19
* thread 2 suma= 19
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ ./a.out
thread 3 suma a[6] / thread 2 suma a[4] / thread 2 suma a[5] / thread 0 suma a[0] / thread 0 suma a[1]
/ thread 1 suma a[2] / thread 1 suma a[3] /
* thread 1 suma= 5
* thread 2 suma= 5
* thread 3 suma= 5
* thread 0 suma= 5
```

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:** Sí, siempre aparecería un 6 si no se cambiase el número de threads, esto se debe al comportamiento que tiene la cláusula `lastprivate`, la cual hace que la variable especificada (además de ser privada para cada thread), tome después de la construcción `parallel` el valor que tendría si se hubiese realizado una ejecución secuencial, (en este caso

como el bucle itera hasta a[6] que es 6, y la hebra 3 no realiza más sumas toma ese valor).

### CAPTURAS DE PANTALLA:

```
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 3 suma a[6] suma=6
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9

Fuera de la construcción parallel suma=6
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ ./a.out
t
thread 3 suma a[6] suma=6
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 0 suma a[0] suma=0
thread 0 suma a[1] suma=1

Fuera de la construcción parallel 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:** El problema que se presenta es que al eliminar `copyprivate` no se realiza la difusión del valor leído por el `scanf` de la hebra que ejecuta el `single` al resto de los threads. En consecuencia el valor de `a` que utilizan el resto de las hebras queda sin inicializar, de ahí que en algunos aparezca el valor 0 y en otro un valor basura (4196990).

### 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 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:

```
Introduce valor de inicialización a: 8

Single ejecutada por el thread 2
Después de la región parallel:
b[0] = 4196990 b[1] = 4196990 b[2] = 4196990 b[3] = 0 b[4] = 0 b[5] = 8 b[6] =
8 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:** Se imprime el mismo resultado que se imprimía con `suma=0` pero ahora incrementado en 10 unidades, esto se debe a que clausula `reduction` junto con el operador `+` va a ir acumulando en la variable `suma` cada una de las sumas parciales que realizan las threads. Esta antes estaba inicializada a 0, por tanto el resultado era la suma, sin embargo ahora al estar inicializada a 10, este valor se tiene en cuenta al realizar la reducción, de ahí que aparezca incrementada.

**CÓDIGO FUENTE:** `reduction-clauseModificado.c`

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

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

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

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

    n = atoi(argv[1]);

    if (n>20){
        n=20;
        printf("n=%d", n);
    }

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

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

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

**CAPTURAS DE PANTALLA:**

```
src: ./a.out
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ gcc reduction-clauseModificado.c -fopenmp
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ ./a.out 8
Tras 'parallel' suma=28
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ gcc reduction-clauseModificado.c -fopenmp
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ ./a.out 8
Tras 'parallel' suma=38
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$
```

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:** Si no vamos a utilizar la clausula `reduction`, obligatoriamente la variable `suma` tiene que ser compartida. Por último, para que las hebras se sincronicen correctamente en el acceso a la variable `suma` se utiliza la directiva `atomic` que no es de trabajo

compartido.

### CÓDIGO FUENTE: reduction-clauseModificado7.c

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

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

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

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

    n = atoi(argv[1]);

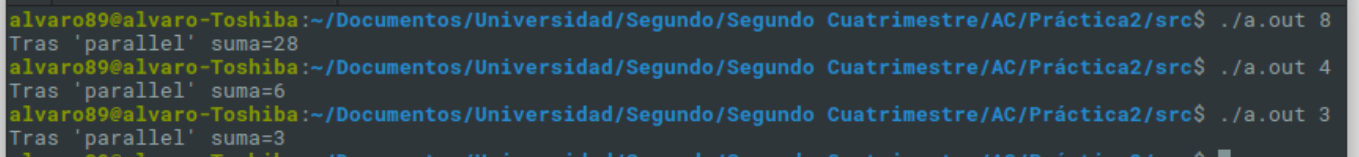
    if (n>20){
        n=20;
        printf("n=%d", n);
    }

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

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

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

### CAPTURAS DE PANTALLA:



```
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ ./a.out 8
Tras 'parallel' suma=28
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ ./a.out 4
Tras 'parallel' suma=6
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$ ./a.out 3
Tras 'parallel' suma=3
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src$
```

### Resto de ejercicios

- 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).

**CÓDIGO FUENTE:** pmv-secuencial.c

```

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

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

    struct timespec cgt1,cgt2;
    double ncgt;

    if(argc < 2){
        printf("Modo de empleo: %s, N\n", argv[0]);
        exit(-1);
    }

    int N = atoi(argv[1]);

    //Reservar memoria dinámica:

    double** M = (double**) malloc(N*sizeof(double));
    for(int i = 0; i<N; ++i)
        M[i] = (double*) malloc(N*sizeof(double));

    double* V = (double*) malloc(N*sizeof(double));

    double* R = (double*) malloc(N*sizeof(double));

    if(M == NULL || V == NULL || R == NULL){
        printf("No se ha podido reservar memoria\n");
        exit(-1);
    }

    //Inicializar elementos:

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

    for(int i = 0; i<N; ++i){
        V[i] = N*0.1 + i*0.1;
        R[i] = 0;
    }

    //Realizar R = M * V y medir tiempo:

    clock_gettime(CLOCK_REALTIME, &cgt1);

    for(int i = 0; i<N; ++i)
        for(int j = 0; j<N; ++j)
            R[i] += (M[i][j] * V[j]);

    clock_gettime(CLOCK_REALTIME, &cgt2);
    ncgt= (double) (cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec -
cgt1.tv_nsec)/(1.e+9));

    //Imprimir resultados:

    if(N<12){
        printf("Tiempo transcurrido (seg): %11.9f \n\nMatriz
generada:\n", ncgt);
        for(int i = 0; i<N; ++i)
            printf("[%0f]", M[0][i]);
        printf("...");
        for(int i = 0; i<N; ++i)
            printf("[%0f]", M[N-1][i]);

        printf("\n\nVector generado:\n");
        for(int i = 0; i<N; ++i)
            printf("[%0f]", V[i]);

        printf("\n\nResultado:\n");
    }
}

```

```

        for(int i = 0; i<N; ++i)
            printf("%.1f", R[i]);

        printf("\n\n");
    }
    else
        printf("Tiempo(seg.): %11.9f \nTamaño: %u \nM[0]
[-]*V=R[0] = %.1f \nM[%d][-]*V=R[%d] = %.1f \n",ncgt,N, R[0], N-1, N-1, R[N-1]);

    //Liberar memoria:
    for(int i = 0; i<N; ++i)
        free(M[i]);

    free(M);
    free(V);
    free(R);

    exit(0);
}

```

**CAPTURAS DE PANTALLA:**

```

alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src
$ gcc pmv-secuencial.c
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src
$ ./a.out 8
Tiempo transcurrido (seg): 0.000001531

Matriz generada:
[0][1][2][3][4][5][6][7]...[7][8][9][10][11][12][13][14]

Vector generado:
[0.8][0.9][1.0][1.1][1.2][1.3][1.4][1.5]

Resultado:
[36.4][45.6][54.8][64.0][73.2][82.4][91.6][100.8]

alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src
$ ./a.out 11
Tiempo transcurrido (seg): 0.000002642

Matriz generada:
[0][1][2][3][4][5][6][7][8][9][10]...[10][11][12][13][14][15][16][17][18][19][20]

Vector generado:
[1.1][1.2][1.3][1.4][1.5][1.6][1.7][1.8][1.9][2.0][2.1]

Resultado:
[99.0][116.6][134.2][151.8][169.4][187.0][204.6][222.2][239.8][257.4][275.0]

```

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).

### CÓDIGO FUENTE : pmv-OpenMP-a.c

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

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

    struct timespec cgt1,cgt2;
    double ncgt;

    if(argc < 2){
        printf("Modo de empleo: %s, N\n", argv[0]);
        exit(-1);
    }

    int N = atoi(argv[1]);

    //Reservar memoria dinámica:

    double** M = (double**) malloc(N*sizeof(double*));
    for(int i = 0; i<N; ++i)
        M[i] = (double*) malloc(N*sizeof(double));

    double* V = (double*) malloc(N*sizeof(double));

    double* R = (double*) malloc(N*sizeof(double));

    if(M == NULL || V == NULL || R == NULL){
        printf("No se ha podido reservar memoria\n");
        exit(-1);
    }

    //Inicializar elementos:

    #pragma omp parallel
    {
        #pragma omp for
        for(int i = 0; i<N; ++i)
            for(int j = 0; j<N; ++j)

                M[i][j] = i+j;

        #pragma omp for
        for(int i = 0; i<N; ++i){
            V[i] = N*0.1 + i*0.1;
            R[i] = 0;
        }
    }

    //Realizar R = M * V y medir tiempo:

    clock_gettime(CLOCK_REALTIME, &cgt1);

    #pragma omp parallel for
    for(int i = 0; i<N; ++i)
        for(int j = 0; j<N; ++j)
            R[i] += (M[i][j] * V[j]);

    clock_gettime(CLOCK_REALTIME, &cgt2);
    ncgt= (double) (cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec -
```

```

cgt1.tv_nsec)/(1.e+9));

    //Imprimir resultados:
    if(N<12){
        printf("Tiempo transcurrido (seg): %11.9f \n\nMatriz
generada:\n", ncgt);
        for(int i = 0; i<N; ++i)
            printf("[%0f]", M[0][i]);
        printf("...");
        for(int i = 0; i<N; ++i)
            printf("[%0f]", M[N-1][i]);

        printf("\n\nVector generado:\n");
        for(int i = 0; i<N; ++i)
            printf("[%0f]", V[i]);

        printf("\n\nResultado:\n");
        for(int i = 0; i<N; ++i)
            printf("[%0f]", R[i]);

        printf("\n\n");
    }
    else
        printf("Tiempo(seg.): %11.9f \nTamaño: %u \nM[0]
[-]*V=R[0] = %0f \nM[%d][-]*V=R[%d] = %0f \n",ncgt,N, R[0], N-1, N-1, R[N-1]);

    //Liberar memoria:
    for(int i = 0; i<N; ++i)
        free(M[i]);

    free(M);
    free(V);
    free(R);

    exit(0);
}

```

**CÓDIGO FUENTE:** pmv-OpenMP-b.c

```

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

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

    struct timespec cgt1,cgt2;
    double ncgt;

    if(argc < 2){
        printf("Modo de empleo: %s, N\n", argv[0]);
        exit(-1);
    }

    int N = atoi(argv[1]);

    //Reservar memoria dinámica:

    double** M = (double**) malloc(N*sizeof(double*));
    for(int i = 0; i<N; ++i)
        M[i] = (double*) malloc(N*sizeof(double));

    double* V = (double*) malloc(N*sizeof(double));

    double* R = (double*) malloc(N*sizeof(double));

    if(M == NULL || V == NULL || R == NULL){
        printf("No se ha podido reservar memoria\n");
    }
}

```

```

        exit(-1);
    }

    //Inicializar elementos:

    #pragma omp parallel
    {
        #pragma omp for
        for(int i = 0; i<N; ++i)
            for(int j = 0; j<N; ++j)
                M[i][j] = i+j;

        #pragma omp for
        for(int i = 0; i<N; ++i){
            V[i] = N*0.1 + i*0.1;
            R[i] = 0;
        }
    }

    //Realizar R = M * V y medir tiempo:

    clock_gettime(CLOCK_REALTIME, &cgt1);

    for(int i = 0; i<N; ++i)
        #pragma omp parallel for
        for(int j = 0; j<N; ++j)
            #pragma omp atomic
            R[i] += (M[i][j] * V[j]);

    clock_gettime(CLOCK_REALTIME, &cgt2);
    ncgt= (double) (cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec -
cgt1.tv_nsec)/(1.e+9));

    //Imprimir resultados:
    if(N<12){
        printf("Tiempo transcurrido (seg): %11.9f \n\nMatriz
generada:\n", ncgt);
        for(int i = 0; i<N; ++i)
            printf("[%0f]", M[0][i]);
        printf("...");
        for(int i = 0; i<N; ++i)
            printf("[%0f]", M[N-1][i]);

        printf("\n\nVector generado:\n");
        for(int i = 0; i<N; ++i)
            printf("[%0f]", V[i]);

        printf("\n\nResultado:\n");
        for(int i = 0; i<N; ++i)
            printf("[%0f]", R[i]);

        printf("\n\n");
    }
    else
        printf("Tiempo(seg.): %11.9f \nTamaño: %u \nM[0]
[-]*V=R[0] = %.1f \nM[%d][-]*V=R[%d] = %.1f \n",ncgt,N, R[0], N-1, N-1, R[N-1]);

    //Liberar memoria:
    for(int i = 0; i<N; ++i)
        free(M[i]);

    free(M);
    free(V);
    free(R);
    exit(0);
}

```

**RESPUESTA:** No he tenido ningún problema con este ejercicio.

### CAPTURAS DE PANTALLA:

```
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Practica2/src
$ gcc pmv-OpenMP-a.c -fopenmp
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Practica2/src
$ ./a.out 8
Tiempo transcurrido (seg): 0.001813565

Matriz generada:
[0][1][2][3][4][5][6][7]...[7][8][9][10][11][12][13][14]

Vector generado:
[0.8][0.9][1.0][1.1][1.2][1.3][1.4][1.5]

Resultado:
[36.4][45.6][54.8][64.0][73.2][82.4][91.6][100.8]

alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Practica2/src
$ ./a.out 11
Tiempo transcurrido (seg): 0.00005479

Matriz generada:
[0][1][2][3][4][5][6][7][8][9][10]...[10][11][12][13][14][15][16][17][18][19][20]

Vector generado:
[1.1][1.2][1.3][1.4][1.5][1.6][1.7][1.8][1.9][2.0][2.1]

Resultado:
[99.0][116.6][134.2][151.8][169.4][187.0][204.6][222.2][239.8][257.4][275.0]
```

```
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Practica2/src
$ gcc pmv-OpenMP-b.c -fopenmp
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Practica2/src
$ ./a.out 8
Tiempo transcurrido (seg): 0.000031366

Matriz generada:
[0][1][2][3][4][5][6][7]...[7][8][9][10][11][12][13][14]

Vector generado:
[0.8][0.9][1.0][1.1][1.2][1.3][1.4][1.5]

Resultado:
[36.4][45.6][54.8][64.0][73.2][82.4][91.6][100.8]

alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Practica2/src
$ ./a.out 11
Tiempo transcurrido (seg): 0.000058577

Matriz generada:
[0][1][2][3][4][5][6][7][8][9][10]...[10][11][12][13][14][15][16][17][18][19][20]

Vector generado:
[1.1][1.2][1.3][1.4][1.5][1.6][1.7][1.8][1.9][2.0][2.1]

Resultado:
[99.0][116.6][134.2][151.8][169.4][187.0][204.6][222.2][239.8][257.4][275.0]
```

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).

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

```

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

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

    struct timespec cgt1,cgt2;
    double ncgt;

    if(argc < 2){
        printf("Modo de empleo: %s, N\n", argv[0]);
        exit(-1);
    }

    int N = atoi(argv[1]);

    //Reservar memoria dinámica:

    double** M = (double**) malloc(N*sizeof(double*));
    for(int i = 0; i<N; ++i)
        M[i] = (double*) malloc(N*sizeof(double));

    double* V = (double*) malloc(N*sizeof(double));

    double* R = (double*) malloc(N*sizeof(double));

    if(M == NULL || V == NULL || R == NULL){
        printf("No se ha podido reservar memoria\n");
        exit(-1);
    }

    //Inicializar elementos:

    #pragma omp parallel
    {
        #pragma omp for
        for(int i = 0; i<N; ++i)
            for(int j = 0; j<N; ++j)

                M[i][j] = i+j;

        #pragma omp for
        for(int i = 0; i<N; ++i)
            V[i] = N*0.1 + i*0.1;
    }

    //Realizar R = M * V y medir tiempo:

    clock_gettime(CLOCK_REALTIME, &cgt1);

    double temp;
    for(int i = 0; i<N; ++i){
        temp = 0;
        #pragma omp parallel for reduction(+:temp)
        for(int j = 0; j<N; ++j)
            temp += (M[i][j] * V[j]);
        R[i] = temp;
    }

    clock_gettime(CLOCK_REALTIME, &cgt2);
    ncgt= (double) (cgt2.tv_sec - cgt1.tv_sec) + (double) ((cgt2.tv_nsec -
cgt1.tv_nsec)/(1.e+9));

    //Imprimir resultados:
    if(N<12){
        printf("Tiempo transcurrido (seg): %11.9f \n\nMatriz
generada:\n", ncgt);
        for(int i = 0; i<N; ++i)

```

```

        printf("%.0f", M[0][i]);
        printf("...");
        for(int i = 0; i<N; ++i)
            printf("%.0f", M[N-1][i]);

        printf("\n\nVector generado:\n");
        for(int i = 0; i<N; ++i)
            printf("%.1f", V[i]);

        printf("\n\nResultado:\n");
        for(int i = 0; i<N; ++i)
            printf("%.1f", R[i]);

        printf("\n\n");
    }
    else
        printf("Tiempo(seg.): %11.9f \nTamaño: %u \nM[0]
[-]*V=R[0] = %.1f \nM[%d][-]*V=R[%d] = %.1f \n",ncgt,N, R[0], N-1, N-1, R[N-1]);

    //Liberar memoria:
    for(int i = 0; i<N; ++i)
        free(M[i]);

    free(M);
    free(V);
    free(R);

    exit(0);
}

```

**RESPUESTA:** No he tenido problemas con este ejercicio.

#### CAPTURAS DE PANTALLA:

```

alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src
$ gcc pmv-OpenMP-reduction.c -fopenmp
alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src
$ ./a.out 8
Tiempo transcurrido (seg): 0.000029704

Matriz generada:
[0][1][2][3][4][5][6][7]...[7][8][9][10][11][12][13][14]

Vector generado:
[0.8][0.9][1.0][1.1][1.2][1.3][1.4][1.5]

Resultado:
[36.4][45.6][54.8][64.0][73.2][82.4][91.6][100.8]

alvaro89@alvaro-Toshiba:~/Documentos/Universidad/Segundo/Segundo Cuatrimestre/AC/Práctica2/src
$ ./a.out 11
Tiempo transcurrido (seg): 0.000042987

Matriz generada:
[0][1][2][3][4][5][6][7][8][9][10]...[10][11][12][13][14][15][16][17][18][19][20]

Vector generado:
[1.1][1.2][1.3][1.4][1.5][1.6][1.7][1.8][1.9][2.0][2.1]

Resultado:
[99.0][116.6][134.2][151.8][169.4][187.0][204.6][222.2][239.8][257.4][275.0]

```

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

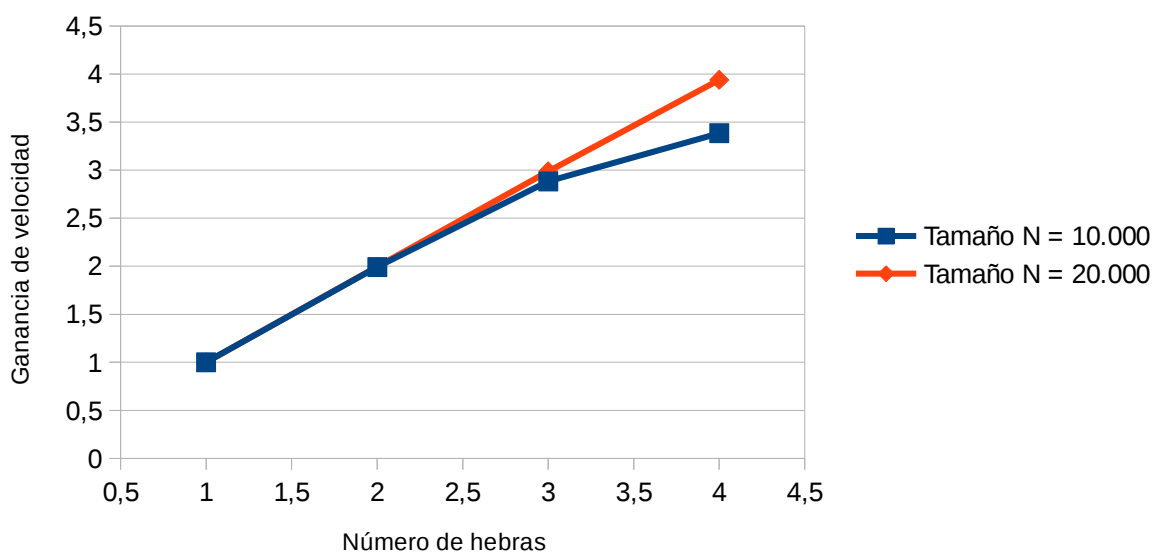
qué el código escogido es el mejor. NOTA: Nunca ejecute en atcgrid código que imprima todos los componentes del resultado.

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

**COMENTARIOS SOBRE LOS RESULTADOS:** Para realizar las mediciones se ha utilizado la versión pmv-OpenMP-a.c ya que me parece la solución más elegante y ligera para el problema del producto de matriz por vector. No requiere comunicaciones extra ni operaciones atómicas como en los otros casos. También he realizado un pequeño estudio empírico de la eficiencia práctica de los tres y este presentaba los mejores resultados. Con respecto a la escalabilidad podemos ver que como se esperaba esta se va incrementando poco a poco. En el PC parece casi lineal pero probablemente si hiciésemos más medidas o con matrices más grandes (lo cual no he podido ya no que soportaba un tamaño mayor) se apreciaría mejor que no es lineal. En atcgrid por su parte este crecimiento se aprecia mejor con un tamaño de 20.000, ya que con 10.000 hay demasiados picos.

<b>Ejercicio 11 (PC LOCAL)</b>			
<b>Nº de Hebras (p)</b>	<b>Tiempo secuencial Ts</b>	<b>Tiempo paralelo para p hebras Tp(p)</b>	<b>Ganancia de velocidad S(p)</b>
<b>Tamaño N = 10.000</b>			
1	0,981889043	0,981889043	1
2	0,981889043	0,493096637	1,9912710194
3	0,981889043	0,340599597	2,8828250287
4	0,981889043	0,290055742	3,3851736091
<b>Tamaño N = 20.000</b>			
1	4,920018522	4,920018522	1
2	4,920018522	2,465939412	1,9951903514
3	4,920018522	1,646667739	2,9878635534
4	4,920018522	1,248748497	3,9399595145

**Escalabilidad PC local**



Ejercicio 11 (ATCGRID)			
Nº de Hebras (p)	Tiempo secuencial Ts	Tiempo paralelo para p hebras Tp(p)	Ganancia de velocidad S(p)
Tamaño N = 10.000			
1	0,147674860	0,147674860	1
2	0,147674860	0,074737761	1,9759069314
3	0,147674860	0,054902077	2,6897863992
4	0,147674860	0,047028186	3,1401351521
5	0,147674860	0,038021639	3,8839688105
6	0,147674860	0,040437494	3,6519290735
7	0,147674860	0,032488994	4,545381122
8	0,147674860	0,03384243	4,3636009589
9	0,147674860	0,050071653	2,9492707181
10	0,147674860	0,032923109	4,485446985
11	0,147674860	0,046630889	3,1668892266
12	0,147674860	0,048768889	3,0280546272
Tamaño N = 20.000			
1	0,651918870	0,651918870	1
2	0,651918870	0,415564344	1,5687555475
3	0,651918870	0,240459327	2,7111398761
4	0,651918870	0,206032726	3,1641520386
5	0,651918870	0,189348943	3,4429496129
6	0,651918870	0,172436628	3,7806287305
7	0,651918870	0,156437805	4,1672719072
8	0,651918870	0,151502445	4,3030254066
9	0,651918870	0,156068505	4,1771327918
10	0,651918870	0,13604446	4,7919545566
11	0,651918870	0,132750624	4,9108535264
12	0,651918870	0,128149898	5,0871587116

### Escalabilidad ATCGRID

