

2º 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): Rafa Nogales

Grupo de prácticas: 1º

Fecha de entrega:

Fecha evaluación en clase:

1. ¿Qué ocurre si en el ejemplo del seminario `shared-clause.c` se añade a la directiva `parallel` la cláusula `default(none)`? Si se plantea algún problema, resuélvalo sin eliminar `default(none)`. Añada el código con la modificación al cuaderno de prácticas.

**RESPUESTA:** Si declaramos “n” como constante no hay ningún problema al compartir variables puesto que las variables constantes no tienen problemas a la hora de sincronizarse entre hebras.

**CÓDIGO FUENTE:** `shared-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=20;

    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:**

```

rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3$ gcc shared-clause.c -o shared_clause -fopenmp
shared-clause.c: En la función 'main':
shared-clause.c:14:12: error: no se especificó 'n' en el paralelo que lo contiene
shared-clause.c:14:12: error: paralelo contenedor
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3$ gcc shared-clause.c -o shared_clause -fopenmp
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3$ ./shared_clause
Después de parallel for:
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
a[6] = 13
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3$ █

```

2. ¿Qué ocurre si en `private-clause.c` se inicializa la variable `suma` fuera de la construcción `parallel` en lugar de dentro? Razone su respuesta. Añada el código con la modificación al cuaderno de prácticas.

**RESPUESTA:** Si inicializamos la variable `suma` fuera del `parallel` no se va a quedar inicializada dentro, pues `private` lo único que hace es crear para cada hebra una variable privada llamada `suma`, pero no le da el valor inicial que tuviese fuera, y por tanto las “sumas” privadas de cada hebra están inicializadas con basurilla.  
(Si utilizamos `-O2` al compilar ninguna hebra se inicializa correctamente).

#### 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;

    suma=0; //Para hacerlo bien se debería borrar esta línea
#pragma omp parallel
    {
        //suma=0 //Para hacerlo bien se debería descomentar esta línea
        #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 de la region Parallel suma= %d", suma);

    printf("\n");
}

```

**CAPTURAS DE PANTALLA:**

```

rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e2$ gcc private-clause.c -o private -fopenmp
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e2$ ./private
thread 1 suma a[2] / thread 1 suma a[3] / 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= 5
* thread 0 suma= 1
* thread 2 suma= 9
* thread 3 suma= 6
* Fuera de la region Parallel suma= -1219132891

```

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

**RESPUESTA:**

Aparece el mismo resultado en todas las hebras porque ahora todas las hebras comparten la misma variable “suma” y el resultado que se muestra es el que tuviese la última hebra en salir del bucle. (Lógicamente este no es el resultado correcto...)

**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=0;

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

    #pragma omp parallel
    {
        #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 de la region Parallel suma= %d", suma);

    printf("\n");
}

```

**CAPTURAS DE PANTALLA:**

```

rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e2$ gcc private-clause.c -o private -fopenmp
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e2$ ./private
thread 0 suma a[0] / thread 1 suma a[2] / thread 1 suma a[3] / thread 0 suma a[1] / thread 3 suma a[6] / thread 2 suma a[4] / thread 2
suma a[5] /
* thread 1 suma= -1081719799
* thread 3 suma= -1081719798
* thread 2 suma= -1081719795
* thread 0 suma= 5
* Fuera de la region Parallel suma= 0
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e2$ gcc -O2 private-clause.c -o private -fopenmp
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e2$ ./private
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= -1217118215
* thread 2 suma= -1217118211
* thread 3 suma= -1217118214
* thread 0 suma= -1080815147
* Fuera de la region Parallel suma= 0
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e2$ █

```

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

**RESPUESTA:** Si, porque la ultima iteración del bucle es la que hace que suma valga 6 y `lastprivate(suma)` hace que la ultima hebra exporte su valor de la variable privada “suma” a la variable externa “suma”.

**CAPTURAS DE PANTALLA:**

```

rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e4$ ls
firstlastprivate firstprivate firstprivate-clause.c firstprivate-clause.c~
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e4$ ./firstprivate
thread 3 suma a[6] suma=6
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

Fuera de la construcción parallel suma=0
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e4$ ./firstlastprivate
thread 2 suma a[4] suma=4
thread 0 suma a[0] suma=0
thread 2 suma a[5] suma=9
thread 0 suma a[1] suma=1
thread 3 suma a[6] suma=6
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5

Fuera de la construcción parallel suma=6
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e4$ ./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 construcción parallel suma=6
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e4$ █

```

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:** La hebra que ejecuta el `single` recibe el valor de “a” y por `copyprivate(a)` lo que hace es difundir el valor de “a” a las otras hebras.

Al eliminar la cláusula `copyprivate(a)` no se produce la difusión y únicamente se inicializan correctamente aquellas zonas que haya inicializado la hebra que ejecutó el `single`.

**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:**

```
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e5$ ./copyprivate-clause
Introduce valor de inicialización a: 3
Single ejecutada por el thread 2
Después de la región parallel:
b[0] = 3      b[1] = 3      b[2] = 3      b[3] = 3      b[4] = 3      b[5] = 3      b[6] = 3      b[7] = 3      b[8] = 3
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e5$ ./copyprivate-clause-mod
Introduce valor de inicialización a: 3
Single ejecutada por el thread 0
Después de la región parallel:
b[0] = 3      b[1] = 3      b[2] = 3      b[3] = -1219216628      b[4] = -1219216628      b[5] = -1219216628      b[6] = -1227609332      b[7] = -1227609332      b[8] = -1227609332
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e5$
```

6. En el ejemplo `reduction-clause.c` sustituya `suma=0` por `suma=10`. ¿Qué resultado se imprime ahora? Justifique el resultado

**RESPUESTA:**

Sale el mismo resultado que con `suma = 0` pero sumándole 10.

Esto se debe a que `reduction` está muy bien implementado y mantiene el valor inicial de la variable de acumulación (`suma` en este caso).

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

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

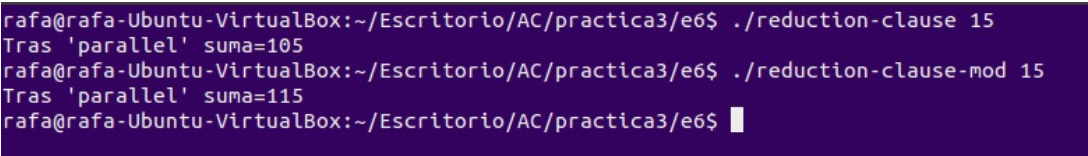
int main(int argc, char **argv) {
    int i, n=20, a[n], suma=10;

    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 += i;
    }

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

**CAPTURAS DE PANTALLA:**


```
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6$ ./reduction-clause 15
Tras 'parallel' suma=105
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6$ ./reduction-clause-mod 15
Tras 'parallel' suma=115
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6$
```

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.



**RESPUESTA:****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=20;

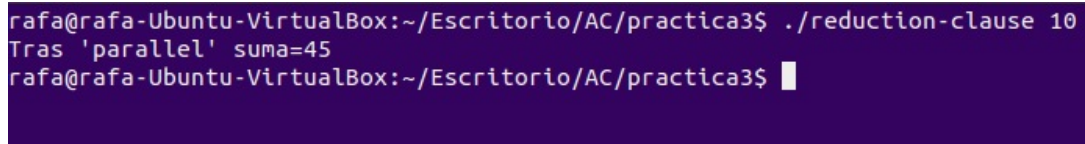
    if(argc < 2) {
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]); if (n>20) {n=20; printf("n=%d",n);}

    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:**


```

rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3$ ./reduction-clause 10
Tras 'parallel' suma=45
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3$

```

8. Implementar un programa secuencial en C que calcule el producto de una matriz cuadrada, M, por un vector, v1:

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

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.cpp

```

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int ** ReservaMatriz(int dim)
{
    int **M;
    if (dim <= 0)
    {
        cerr<< "\n ERROR: Dimension de la matriz debe ser mayor que 0" <<
endl;
        exit(1);
    }
    M = new int* [dim];
    if (M == NULL)
    {
        cerr << "\n ERROR: No puedo reservar memoria para un matriz de "
<< dim << " x " << dim << "elementos" << endl;
        exit(1);
    }
    for (int i = 0; i < dim; i++)
    {
        M[i]= new int [dim];
        if (M[i] == NULL)
        {
            cerr << "ERROR: No puedo reservar memoria para un matriz de "
<< dim << " x " << dim << endl;
            for (int j = 0; j < i; j++)
                delete [] M[j];
            delete [] M;
            exit(1);
        }
    }
    return M;
}

void LiberaMatriz(int ** & M, int dim)
{
    for (int i = 0; i < dim; i++)
        delete [] M[i];
    delete [] M;
    M = NULL;
}

void LiberaVector(int * & V)
{
    delete [] V;
    V = NULL;
}

void MultiplicarMatrices(int** & Matriz, int* & vector, int* & resultado, int
dim){
    for(int i=0; i<dim; i++){
        resultado[i]=0;
        for(int j=0; j<dim; j++){
            resultado[i] += Matriz[i][j] * vector[j];
        }
    }
}

```



```

}

void ImprimirProducto(int** & matriz, int* & vector, int* & vector_resultante,
int dim){
    for(int i=0; i<dim; i++){
        for(int j=-1; j<dim+6; j++){
            if(j==-1){
                cout << "( ";
            }
            if(0<=j && j<dim){
                cout << matriz[i][j] << " ";
            }
            else if(j==dim){
                cout << ") (";
            }
            else if(j==dim+1){
                cout << vector[i];
            }
            else if(j==dim+2){
                cout << ") ";
            }
            else if(j==dim+3){
                if(i == dim/2){
                    cout << " = (";
                }
                else
                    cout << "    (";
            }
            else if(j==dim+4){
                cout << vector_resultante[i];
            }
            else if(j==dim+5){
                cout << ")";
            }
        }
        cout << endl;
    }
}

int main(int argc, char **argv){
    int N;
    clock_t tantes;    // Valor del reloj antes de la ejecucion
    clock_t tdespues;  // Valor del reloj despues de la ejecucion

    if(argc < 2){
        cerr << "Falta iteraciones" << endl;
        return(-1);
    }
    N = atoi(argv[1]);

    //Reserva de memoria para los vectores(N) y la matriz(NxN)
    int * vector = new int[N];
    int * vector_resultante = new int[N];
    int ** matriz = ReservaMatriz(N);
    //Inicializar vector
    for (int i=0; i < N; i++){
        vector[i] = 1;
    }
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            matriz[i][j] = i+j;

```

```

    }
}
//int suma=0;
tantes = clock();
MultiplicarMatrices(matriz, vector, vector_resultante, N);
tdespues = clock();
if(N<10){
    ImprimirProducto(matriz, vector, vector_resultante, N);
    cout << endl;
}
cout <<"Tiempo: " << ((double)(tdespues-tantes))/CLOCKS_PER_SEC << " s" <<
endl;
LiberaMatriz(matriz,N);
LiberaVector(vector);
LiberaVector(vector_resultante);
}

```

**CAPTURAS DE PANTALLA:**

No pongo las capturas desde Ubuntu porque se me ha vuelto un poco loco el VirtualBox después de actualizarlo.

```

[ElEstudiante16@atcgrid practica2]$ ./secuencial 5
( 0 1 2 3 4 ) (1)   (10)
( 1 2 3 4 5 ) (1)   (15)
( 2 3 4 5 6 ) (1) = (20)
( 3 4 5 6 7 ) (1)   (25)
( 4 5 6 7 8 ) (1)   (30)

Tiempo: 0 s
[ElEstudiante16@atcgrid practica2]$ ./secuencial 1000
Tiempo: 0 s
[ElEstudiante16@atcgrid practica2]$ ./secuencial 10000
Tiempo: 0.24 s
[ElEstudiante16@atcgrid practica2]$ █

```

9. Implementar en paralelo el producto matriz por vector con OpenMP a partir del código escrito en el ejercicio anterior. 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 <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>
using namespace std;

int ** ReservaMatriz(int dim)
{
    int **M;
    if (dim <= 0)
    {
        cerr<< "\n ERROR: Dimension de la matriz debe ser mayor que 0" <<
endl;
        exit(1);
    }
    M = new int* [dim];
    if (M == NULL)
    {
        cerr << "\n ERROR: No puedo reservar memoria para un matriz de "
<< dim << " x " << dim << "elementos" << endl;
        exit(1);
    }
    for (int i = 0; i < dim; i++)
    {
        M[i]= new int [dim];
        if (M[i] == NULL)
        {
            cerr << "ERROR: No puedo reservar memoria para un matriz de "
<< dim << " x " << dim << endl;
            for (int j = 0; j < i; j++)
                delete [] M[j];
            delete [] M;
        }
    }
}
```

```

        exit(1);
    }
}
return M;
}

void LiberaMatriz(int ** & M, int dim)
{
    for (int i = 0; i < dim; i++)
        delete [] M[i];
    delete [] M;
    M = NULL;
}

void LiberaVector(int * & V)
{
    delete [] V;
    V = NULL;
}

void MultiplicarMatrices(int** & Matriz, int* & vector, int* & resultado, int
dim){
    #pragma omp parallel for
    for(int i=0; i<dim; i++){
        resultado[i]=0;
        for(int j=0; j<dim; j++){
            resultado[i] += Matriz[i][j] * vector[j];
        }
    }
    cout << resultado[0] << " " << resultado[dim-1] << endl;
}

void ImprimirProducto(int** & matriz, int* & vector, int* & vector_resultante,
int dim){
    for(int i=0; i<dim; i++){
        for(int j=-1; j<dim+6; j++){
            if(j==-1){
                cout << "(" << " ";
            }
            if(0<=j && j<dim){
                cout << matriz[i][j] << " ";
            }
            else if(j==dim){
                cout << ")" << " ";
            }
            else if(j==dim+1){
                cout << vector[i];
            }
            else if(j==dim+2){
                cout << " ";
            }
            else if(j==dim+3){
                if(i == dim/2){
                    cout << " = (" << " ";
                }
                else
                    cout << " (" << " ";
            }
            else if(j==dim+4){
                cout << vector_resultante[i];
            }
        }
    }
}

```

```

        else if(j==dim+5){
            cout << " ";
        }
    }
    cout << endl;
}
}

int main(int argc, char **argv){
    int N;
    double tantes;    // Valor del reloj antes de la ejecucion
    double tdespues;  // Valor del reloj despues de la ejecucion

    if(argc < 2){
        cerr << "Falta iteraciones" << endl;
        return(-1);
    }
    N = atoi(argv[1]);

    //Reserva de memoria para los vectores(N) y la matriz(NxN)
    int * vector = new int[N];
    int * vector_resultante = new int[N];
    int ** matriz = ReservaMatriz(N);
    //Inicializar vector
    for (int i=0; i < N; i++){
        vector[i] = 1;
    }
    for(int i=0; i<N; i++){
        for(int j=0; j<N; j++){
            matriz[i][j] = i+j;
        }
    }
    //int suma=0;
    tantes = omp_get_wtime();
    MultiplicarMatrices(matriz, vector, vector_resultante, N);
    tdespues = omp_get_wtime();
    if(N<10){
        ImprimirProducto(matriz, vector, vector_resultante, N);
        cout << endl;
    }
    cout <<"Tiempo: " << (tdespues-tantes) << " s" << endl;
    LiberaMatriz(matriz,N);
    LiberaVector(vector);
    LiberaVector(vector_resultante);
}

```

### CÓDIGO FUENTE: pmv-OpenMP-b.c

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>
using namespace std;

int ** ReservaMatriz(int dim){ ... }
void LiberaMatriz(int ** & M, int dim) { ... }
void LiberaVector(int * & V) { ... }
void ImprimirProducto(int**& matriz, int*& vector, int*& vector_resultante,
int dim) {...}

void MultiplicarMatrices(int** & Matriz, int* & vector, int* & resultado, int
dim){
    int aux_suma=0;
    for(int i=0; i<dim; i++){
        #pragma omp parallel
        {
            int coordenadaIesima_privada=0;
            #pragma omp for
            for (int j=0; j<dim; j++){
                coordenadaIesima_privada
+= Matriz[i][j] * vector[j];
            }
            #pragma omp atomic
            resultado[i] +=
coordenadaIesima_privada;
        }
    }
    cout << resultado[0] << " " << resultado[dim-1] << endl;
}

int main(int argc, char **argv){
//El mismo main que antes
}
```

### RESPUESTA:

### CAPTURAS DE PANTALLA:

```
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6/multiplicaMatrices$ ./multparallelcols 1000
499500 1498500
Tiempo: 0.0018992 s
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6/multiplicaMatrices$ ./multparallelcols 1000
499500 1498500
Tiempo: 0.00363249 s
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6/multiplicaMatrices$ █
```

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 <iostream>
#include <cstdlib>
#include <ctime>
#include <omp.h>
using namespace std;

int ** ReservaMatriz(int dim)
{
    ...
}
void LiberaMatriz(int ** & M, int dim) {
    ...
}
void LiberaVector(int * & V)
{
    ...
}

void ImprimirProducto(int**& matriz, int*& vector, int*& vector_resultante,
int dim) {...}

void MultiplicarMatrices(int** & Matriz, int* & vector, int* & resultado, int
dim){
    for(int i=0; i<dim; i++){
        resultado[i]=0;
        int acumulado=0;
        #pragma omp parallel for reduction(+:acumulado)
        for(int j=0; j<dim; j++){
            acumulado += Matriz[i][j] * vector[j];
        }
        resultado[i] = acumulado;
    }
    cout << resultado[0] << " " << resultado[dim-1] << endl;
}

int main(int argc, char **argv){
    //El mismo que en los otros dos
}
```



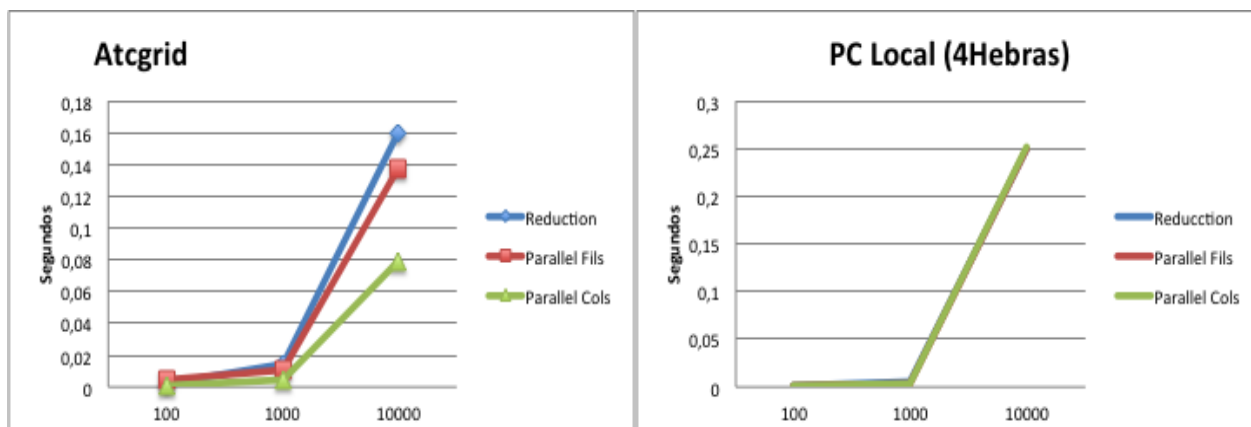
**CAPTURAS DE PANTALLA:**

```

rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6/multiplicaMatrices$ ./multReduction 1000
499500 1498500
Tiempo: 0.00365608 s
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6/multiplicaMatrices$ ./multparallelfils 1000
499500 1498500
Tiempo: 0.0018992 s
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6/multiplicaMatrices$ ./multparallelcols 1000
499500 1498500
Tiempo: 0.00363249 s
rafa@rafa-Ubuntu-VirtualBox:~/Escritorio/AC/practica3/e6/multiplicaMatrices$ █

```

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 del aula de prácticas de los tres códigos implementados en los ejercicios anteriores para tres tamaños (N) distintos (consulte la Lección 6/Tema 2). Usar `-O2` al compilar.



(Los datos están tomados sin la opción `-O2`)

**PC-Local**

Tamaño	Reduction	Paralelo Fils	Paralelo Cols
100	0,00115713	0,000692826	0,000525749
1000	0,00458438	0,00399967	0,00380238
10000	0,249109	0,249109	0,250993

**Atcgrid**

Tamaño	Reduction	Paralelo Fils	Paralelo Cols
100	0,00205908	0,00409402	0,00088389
1000	0,0147584	0,0108157	0,00503124
10000	0,159794	0,137681	0,0793289

**COMENTARIOS SOBRE LOS RESULTADOS:** Va mas rápido en secuencial porque los ejemplos son de juguete, para tamaños de 100.000 hace falta más core así que no puede notarse la potencia del cálculo paralelo, porque se tarda más en crear las hebras que en lo que van a calcular...