

Arquitectura de Computadores (AC)

Cuaderno de prácticas.

Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Rodrigo Raya Castellano

Grupo de prácticas: 2

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)? (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:

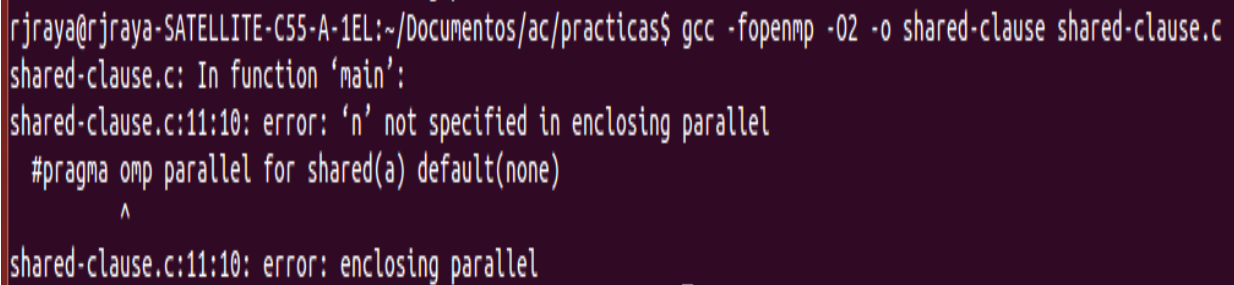
Se produce un error que nos informa de que no se ha definido el alcance de la variable n cuando la directiva default(none) exige que todos los ámbitos los defina el programador.

Para solucionarlo, añadimos shared(n) de modo que n sea una variable compartida por todos los threads.

CÓDIGO FUENTE: shared-clauseModificado.c

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
main()
{
    int i, n = 7;
    int a[n];
    for (i=0; i<n; i++)
        a[i] = i+1;
    #pragma omp parallel for shared(a) shared(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:



```
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas$ gcc -fopenmp -O2 -o shared-clause shared-clause.c
shared-clause.c: In function 'main':
shared-clause.c:11:10: error: 'n' not specified in enclosing parallel
    #pragma omp parallel for shared(a) default(none)
    ^
shared-clause.c:11:10: 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? Razone su respuesta. Añada el código con la modificación al cuaderno de prácticas.

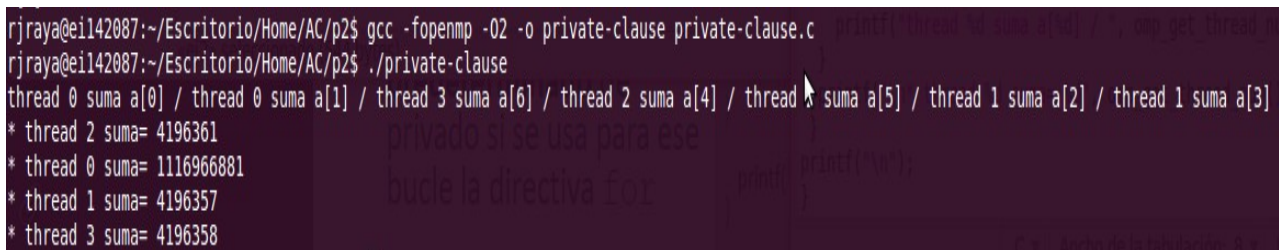
RESPUESTA:

El valor de entrada estará indefinido y las diferentes copias de la variable `suma` contienen basura, luego no se imprime el valor correcto.

CÓDIGO FUENTE: `private-clauseModificado.c`

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif
main()
{
    int i, n = 7;
    int a[n], suma = 0;
    for (i=0; i<n; i++)
        a[i] = i;
    #pragma omp parallel private(suma)
    {
        #pragma omp for
        for (i=0; i<n; i++)
        {
            suma = suma + a[i];
            printf("thread %d suma a[%d] / ", omp_get_thread_num(), i);
        }
        printf("\n* thread %d suma= %d", omp_get_thread_num(), suma);
    }
    printf("\n");
}
```

CAPTURAS DE PANTALLA:



```
rjraya@ei142087:~/Escritorio/Home/AC/p2$ gcc -fopenmp -O2 -o private-clause private-clause.c
rjraya@ei142087:~/Escritorio/Home/AC/p2$ ./private-clause
thread 0 suma a[0] / thread 0 suma a[1] / thread 3 suma a[6] / thread 2 suma a[4] / thread 1 suma a[5] / thread 1 suma a[2] / thread 1 suma a[3]
* thread 2 suma= 4196361
* thread 0 suma= 1116966881
* thread 1 suma= 4196357
* thread 3 suma= 4196358
```

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

RESPUESTA:

Todos los threads comparten la misma variable suma y por tanto al final se imprime la misma. También observamos que variando el número de threads ,varía la suma que se obtiene.

CÓDIGO FUENTE: private-clauseModificado3.c

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

```
rjraya@eil42087:~/Escritorio/Home/AC/p2$ export OMP_NUM_THREADS=6
rjraya@eil42087:~/Escritorio/Home/AC/p2$ ./private_clause
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 2 suma= 3
* thread 1 suma= 3
* thread 4 suma= 3
* thread 0 suma= 3
* thread 3 suma= 3
* thread 5 suma= 3
rjraya@eil42087:~/Escritorio/Home/AC/p2$ export OMP_NUM_THREADS=8
rjraya@eil42087:~/Escritorio/Home/AC/p2$ ./private_clause
thread 3 suma a[3] / thread 2 suma a[2] / thread 0 suma a[0] / thread 6 suma a[6] / thread 5 suma a[5] / thread 1 suma a[1] / thread 4 suma a[4]
* thread 7 suma= 0
* thread 4 suma= 0
* thread 1 suma= 0
* thread 0 suma= 0
* thread 6 suma= 0
* thread 2 suma= 0
* thread 5 suma= 0
* thread 3 suma= 0
```

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:

La cláusula devuelve el valor en la última iteración del bucle. Como `a[6] = 6` devuelve 6. Si cambiara el número de iteraciones y el tamaño del array entonces cambiaría el resultado.

CAPTURAS DE PANTALLA:

```
rjraya@eil42087:~/Escritorio/Home/AC/p2$ ./firstlast-private
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
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5

Fuera de la construcción parallel suma=6
rjraya@eil42087:~/Escritorio/Home/AC/p2$ ./firstlast-private
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5

Fuera de la construcción parallel suma=6
```

Con otro número de iteraciones:

```
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas$ gcc -fopenmp -O2 -o
firstlast-private firstlast-private.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas$ ./firstlast-private
thread 2 suma a[6] suma=6
thread 2 suma a[7] suma=13
thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 0 suma a[2] suma=3
thread 3 suma a[8] suma=8
thread 3 suma a[9] suma=17
thread 1 suma a[3] suma=3
thread 1 suma a[4] suma=7
thread 1 suma a[5] suma=12

Fuera de la construcción parallel suma=17
```

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:

Algunas iteraciones (las de hebra que ejecuta el `single`) obtienen el valor correcto y las demás un valor basura. La razón es que no se les hace llegar (no se difunde) el valor de la variable leída.

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:

```
rjraya@eil42087:~/Escritorio/Home/AC/p2$ gcc -fopenmp -o copyprivate-clause copyprivate-clause.c
rjraya@eil42087:~/Escritorio/Home/AC/p2$ ./copyprivate-clause

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] = 3    b[4] = 3    b[5] = 3    b[6] = 3    b[7] = 3    b[8] = 3
rjraya@eil42087:~/Escritorio/Home/AC/p2$ gcc -fopenmp -o copyprivate-clause copyprivate-clause.c
rjraya@eil42087:~/Escritorio/Home/AC/p2$ ./copyprivate-clause

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] = 141943589    b[4] = 141943589    b[5] = 141943589    b[6] = 0    b[7] = 0    b[8] = 0
rjraya@eil42087:~/Escritorio/Home/AC/p2$
```

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

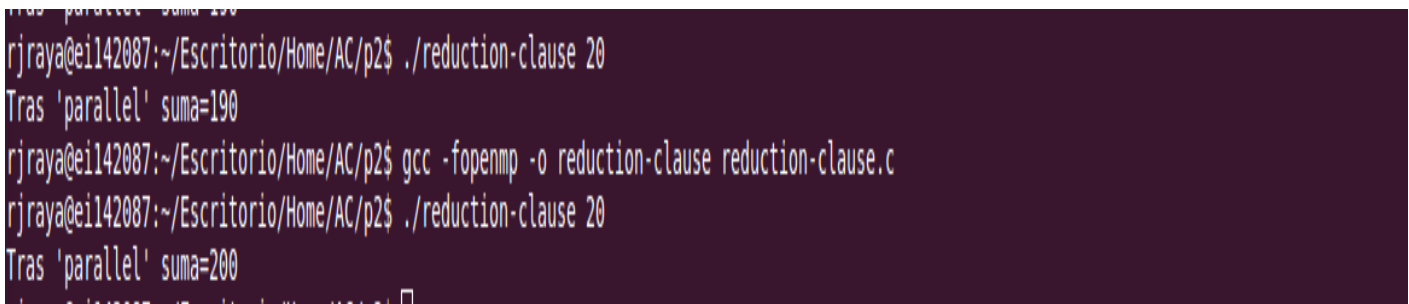
RESPUESTA:

Con 20 iteraciones y con `suma = 0` imprime 190 y con `suma = 10` imprime 200 . Esto ocurre así porque también se reduce la variable compartida.

CÓDIGO FUENTE: `reduction-clauseModificado.c`

```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif
main(int argc, char **argv) {
    int i, n=20, a[n], suma=10;
    if(argc < 2) {
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]); if (n>20) {n=20; printf("n=%d",n);}
    for (i=0; i<n; i++) a[i] = i;
    #pragma omp parallel for reduction(+:suma)
    for (i=0; i<n; i++) suma += a[i];
    printf("Tras 'parallel' suma=%d\n", suma);
}
```

CAPTURAS DE PANTALLA:



```
rjraya@ei142087:~/Escritorio/Home/AC/p2$ ./reduction-clause 20
Tras 'parallel' suma=190
rjraya@ei142087:~/Escritorio/Home/AC/p2$ gcc -fopenmp -o reduction-clause reduction-clause.c
rjraya@ei142087:~/Escritorio/Home/AC/p2$ ./reduction-clause 20
Tras 'parallel' suma=200
```

7. En el ejemplo `reduction-clause.c`, elimine `for` 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:

Lo que he hecho ha sido repartir explícitamente las iteraciones del bucle en modo round-robin.

CÓDIGO FUENTE: `reduction-clauseModificado7.c`

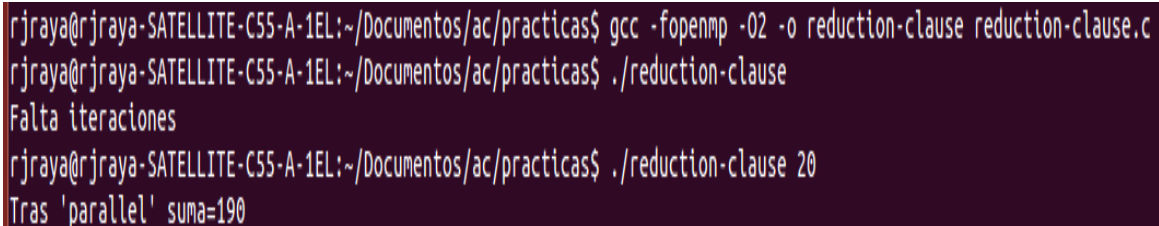
```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif
main(int argc, char **argv) {
    int i, n=20, a[n], suma=0;
    if(argc < 2) {
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]); if (n>20) {n=20; printf("n=%d", n);}

    int tid = -1;

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

    #pragma omp parallel reduction(+:suma) private(tid)
    {
        int nT = omp_get_num_threads();
        tid = omp_get_thread_num();
        for (i=tid; i<n; i = i + nT){
            suma += a[i];
        }
    }
    printf("Tras 'parallel' suma=%d\n", suma);
}
```

CAPTURAS DE PANTALLA:



```
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas$ gcc -fopenmp -O2 -o reduction-clause reduction-clause.c
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas$ ./reduction-clause
Falta iteraciones
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas$ ./reduction-clause 20
Tras 'parallel' suma=190
```

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

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

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.c

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

void main(int argc, char **argv) {
    int N=atoi(argv[1]), i, j;
    double t1,t2;
    int *V;
    V = (int *)malloc (N*sizeof(int));
    int *V2;
    V2 = (int *)malloc (N*sizeof(int));
    int **M;
    M = (int **)malloc (N*sizeof(int *));
    for(i=0; i<N; i++)
        M[i] = (int *)malloc (N*sizeof(int));

    //Comprobamos que se dieron bien los parámetros
    if(argc < 2) {
        fprintf(stderr, "Uso: ./ejercicio8 N \n");
        exit(-1);
    }
    srand(time(0));

    //Inicializamos M y V
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
            M[i][j] = 1;

    for(i=0; i<N; i++)
        V[i] = i+1;

    //Multiplicamos M x V = V2
    t1 = omp_get_wtime();
    for(i=0; i<N; i++){
        V2[i] = 0;
        for(j=0; j<N; j++)
            V2[i] += M[i][j] * V[j];
    }
    t2 = omp_get_wtime();
    t2 = t2-t1;

    //Mostrar los resultados
```



```

if(N<10){
    printf("M:\n");
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            printf("%d ",M[i][j]);
            printf("\n");
        }
        printf("\nV:\n");
        for(i=0; i<N; i++) printf("%d \n",V[i]);
        printf("\nResultado MxV:\n");
        for(i=0; i<N; i++) printf("%d ",V2[i]);
    }
    else{
        printf("V2[0]:%d\n",V2[0]);
        printf("V2[N-1]:%d\n",V2[N-1]);
    }
    printf("\nTiempo en ejecutar MxV:%8.7f\n",t2);

//Liberar memoria

    for(i=0; i<N; i++)
        free(M[i]);
    free(M);
    free(V);
    free(V2);
}

```

CAPTURAS DE PANTALLA:

```

rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico2$ ./ej8 8
M:
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1

V:
1
2
3
4
5
6
7
8

Resultado MxV:
36 36 36 36 36 36 36 36
Tiempo en ejecutar MxV:0.0000005

```

```

rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico2$ ./ej8 11
V2[0]:66
V2[N-1]:66

Tiempo en ejecutar MxV:0.0000007

```

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

void main(int argc, char **argv) {
    int N=atoi(argv[1]), i, j;
    double t1,t2;
    int *V;
    V = (int *)malloc (N*sizeof(int));
    int *V2;
    V2 = (int *)malloc (N*sizeof(int));
    int **M;
    M = (int **)malloc (N*sizeof(int *));
    for(i=0; i<N; i++)
        M[i] = (int *)malloc (N*sizeof(int));

    //Comprobamos que se dieron bien los parámetros
    if(argc < 2) {
        fprintf(stderr, "Uso: ./ejercicio9 N \n");
        exit(-1);
    }
    srand(time(0));

    //Inicializamos M y V
    #pragma omp parallel for private(j)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            M[i][j] = 1;
        }
    }
}
```

```

    }
}

#pragma omp parallel for
for(i=0; i<N; i++)
    V[i] = i+1;

//Multiplicamos M x V = V2
t1 = omp_get_wtime();
int res ;
#pragma omp parallel private(res,j)
{
    #pragma omp for
    for(i=0; i<N; i++){
        res = 0;
        for(j=0; j<N; j++){
            res += M[i][j] * V[j];
        }
        V2[i] = res;
    }
}

t2 = omp_get_wtime();
t2 = t2-t1;

//Mostrar los resultados
if(N<10){
    printf("M:\n");
    for(i=0; i<N; i++){
        for(j=0; j<N; j++)
            printf("%d ",M[i][j]);
        printf("\n");
    }
    printf("\nV:\n");
    for(i=0; i<N; i++) printf("%d \n",V[i]);
    printf("\nResultado MxV:\n");
    for(i=0; i<N; i++) printf("%d ",V2[i]);
}
else{
    printf("V2[0]:%d\n",V2[0]);
    printf("V2[N-1]:%d\n",V2[N-1]);
}
printf("\nTiempo en ejecutar MxV:%8.7f\n",t2);

//Liberar memoria

for(i=0; i<N; i++)
    free(M[i]);
free(M);
free(V);
free(V2);
}

```

CÓDIGO FUENTE: pmv-OpenMP-b.c

```

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

```

```

void main(int argc, char **argv) {
    int N=atoi(argv[1]), i, j;
    double t1,t2;
    int *V;
    V = (int *)malloc (N*sizeof(int));
    int *V2;
    V2 = (int *)malloc (N*sizeof(int));
    int **M;
    M = (int **)malloc (N*sizeof(int *));
    for(i=0; i<N; i++)
        M[i] = (int *)malloc (N*sizeof(int));

    //Comprobamos que se dieron bien los parámetros
    if(argc < 2) {
        fprintf(stderr, "Uso: ./ejercicio9 N \n");
        exit(-1);
    }
    srand(time(0));

    //Inicializamos M y V
    #pragma omp parallel for private(j)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            M[i][j] = 1;
        }
    }

    #pragma omp parallel for
    for(i=0; i<N; i++)
        V[i] = i+1;

    //Multiplicamos M x V = V2
    t1 = omp_get_wtime();
    int suma,sumalocal;

    for(i=0; i<N; i++){
        suma = 0;
        #pragma omp parallel private(sumalocal)
        {
            sumalocal = 0;
            #pragma omp for
            for(j=0; j<N; j++){
                sumalocal += M[i][j] * V[j];
            }
            #pragma omp critical
            suma += sumalocal;
        }

        V2[i] = suma;
    }

    t2 = omp_get_wtime();
    t2 = t2-t1;

    //Mostrar los resultados
    if(N<10){
        printf("M:\n");
        for(i=0; i<N; i++){
            for(j=0; j<N; j++)

```

```

        printf("%d ",M[i][j]);
        printf("\n");
    }
    printf("\nV:\n");
    for(i=0; i<N; i++) printf("%d \n",V[i]);
    printf("\nResultado MxV:\n");
    for(i=0; i<N; i++) printf("%d ",V2[i]);
}
else{
    printf("V2[0]:%d\n",V2[0]);
    printf("V2[N-1]:%d\n",V2[N-1]);
}
printf("\nTiempo en ejecutar MxV:%8.7f\n",t2);

//Liberar memoria

for(i=0; i<N; i++)
    free(M[i]);
free(M);
free(V);
free(V2);
}

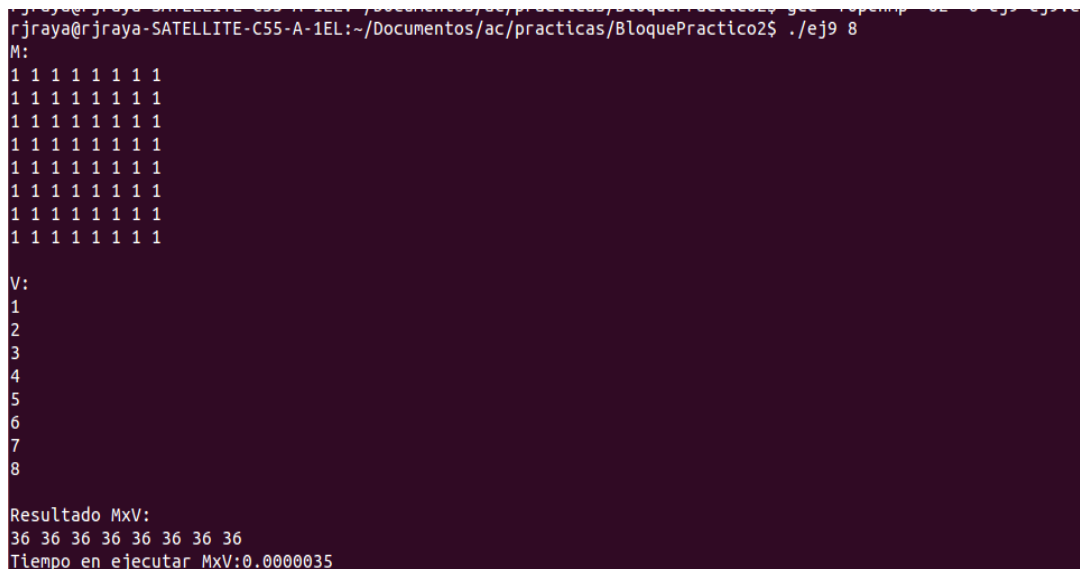
```

RESPUESTA:

Los fallos de compilación los obtuve sobre todo en el ejercicio 8 a la hora de reservar memoria y liberar memoria en C y respecto a los formatos para printf. He usado diversos foros de internet para “recordar” algunas de estas sutilezas así como la ayuda de mis compañeros.

Sí que tuve problemas con fallos en tiempo de ejecución. En la primera versión olvidé que la variable j debía ser privada en el bucle más interno (en el externo ya lo es porque la afecta la directiva for). Esto provocaba que la matriz no quedara bien inicializada. Se obtiene un resultado diferente en cada ejecución.

En el segundo apartado tuve que hacer varios intentos hasta darme cuenta que necesitaba una sumalocal y una suma general que solo pudiera modificarse en una sección crítica. Este problema lo resolví sin ayuda.

CAPTURAS DE PANTALLA:


```

rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico2$ ./ej9 8
M:
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1

V:
1
2
3
4
5
6
7
8

Resultado MxV:
36 36 36 36 36 36 36 36
Tiempo en ejecutar MxV:0.0000035

```

```

rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico2$ ./ej9 11
V2[0]:66
V2[N-1]:66

Tiempo en ejecutar MxV:0.0000036

```

Las imágenes del apartado b) son idénticas a las de este apartado.

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

void main(int argc, char **argv) {
    int N=atoi(argv[1]), i, j;
    double t1,t2;
    int *V;
    V = (int *)malloc (N*sizeof(int));
    int *V2;
    V2 = (int *)malloc (N*sizeof(int));
    int **M;
    M = (int **)malloc (N*sizeof(int *));
    for(i=0; i<N; i++)
        M[i] = (int *)malloc (N*sizeof(int));

    //Comprobamos que se dieron bien los parámetros
    if(argc < 2) {
        fprintf(stderr, "Uso: ./ejercicio9 N \n");
        exit(-1);
    }
    srand(time(0));

    //Inicializamos M y V
    #pragma omp parallel for private(j)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++){
            M[i][j] = 1;
        }
    }

    #pragma omp parallel for
    for(i=0; i<N; i++)
        V[i] = i+1;

```

```

//Multiplicamos M x V = V2
t1 = omp_get_wtime();
int suma;

for(i=0; i<N; i++){
    suma = 0; //suma compartida
    #pragma omp parallel
    {
        #pragma omp for reduction(+:suma)
        for(j=0; j<N; j++)
            suma += M[i][j] * V[j];
    }
    V2[i] = suma;
}

t2 = omp_get_wtime();
t2 = t2-t1;

//Mostrar los resultados
if(N<10){
    printf("M:\n");
    for(i=0; i<N; i++){
        for(j=0; j<N; j++)
            printf("%d ",M[i][j]);
        printf("\n");
    }
    printf("\nV:\n");
    for(i=0; i<N; i++) printf("%d \n",V[i]);
    printf("\nResultado MxV:\n");
    for(i=0; i<N; i++) printf("%d ",V2[i]);
}
else{
    printf("V2[0]:%d\n",V2[0]);
    printf("V2[N-1]:%d\n",V2[N-1]);
}
printf("\nTiempo en ejecutar MxV:%8.7f\n",t2);

//Liberar memoria

for(i=0; i<N; i++)
    free(M[i]);
free(M);
free(V);
free(V2);
}

```

RESPUESTA:

En tiempo de compilación he tenido problemas al principio pues mantenía la doble nomenclatura para sumalocal y suma pero no me ha supuesto una gran dificultad.

En tiempo de ejecución he corregido una definición del índice del primer bucle iterador como privado (esto no causaba problemas).

CAPTURAS DE PANTALLA:

```
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico2$ time ./ej10 8
M:
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1

V:
1
2
3
4
5
6
7
8

Resultado MxV:
36 36 36 36 36 36 36 36
Tiempo en ejecutar MxV:0.0000142

real    0m0.002s
user    0m0.000s
sys     0m0.003s
```

```
rjraya@rjraya-SATELLITE-C55-A-1EL:~/Documentos/ac/practicas/BloquePractico2$ time ./ej10 11
V2[0]:66
V2[N-1]:66

Tiempo en ejecutar MxV:0.0000241

real    0m0.002s
user    0m0.002s
sys     0m0.001s
```


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.

TABLA Y GRÁFICA (por ejemplo para 1-4 threads PC aula, y para 1-12 threads en atcgrid, tamaños-N-: 1.000, 10.000, 100.000):

A continuación muestro los resultados de ejecución en mi ordenador personal para 2,3 y 4 threads y tamaños 1000, 5000, y 10000.

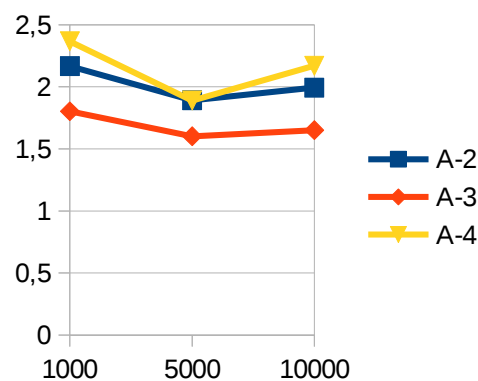
Tiempos:

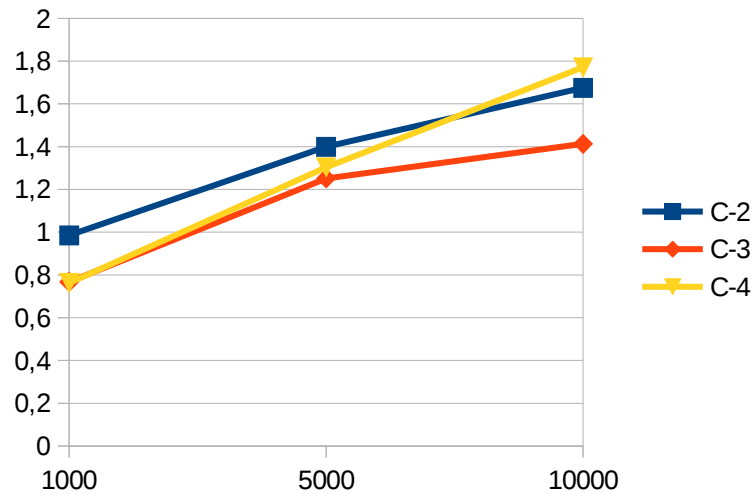
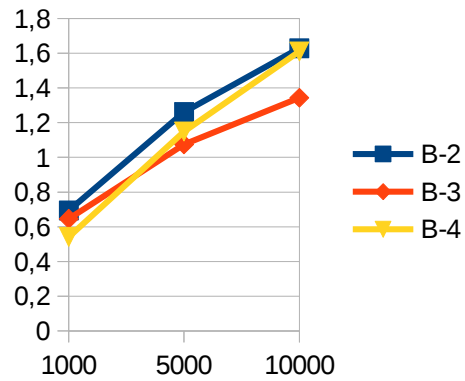
	Secuencial	Paralelo-A			Paralelo-B			Paralelo-C		
	1	2	3	4	2	3	4	2	3	4
1000	0,002082	0,000961	0,0011547	0,0008798	0,0029998	0,0032203	0,0038432	0,0021141	0,0027124	0,0027299
5000	0,0314055	0,0166007	0,0195991	0,0166178	0,024914	0,0292372	0,0273359	0,0224515	0,0250945	0,0240935
10000	0,0936637	0,0469781	0,0567176	0,0431643	0,0575	0,0697186	0,0581736	0,0559324	0,0662831	0,0528984

Ganancias:

	Ganancia-A			Ganancia-B			Ganancia-C		
Tamaño	A-2	A-3	A-4	B-2	B-3	B-4	C-2	C-3	C-4
1000	2,1664932362	1,8030657314	2,3664469198	0,6940462698	0,6465236158	0,5417360533	0,9848162339	0,7675859018	0,7626652991
5000	1,8918178149	1,6023950079	1,8898711021	1,2605563137	1,0741623685	1,1488738253	1,3988152239	1,2514893702	1,3034843422
10000	1,9937736946	1,6514045023	2,1699344134	1,628933913	1,3434535404	1,6100722665	1,6745875378	1,4130856885	1,7706338944

Gráficas que representan la ganancia en función del número de threads (códigos A,B y C)





A continuación presento los resultados obtenidos en atcgrid:

Tabla general de tiempos:

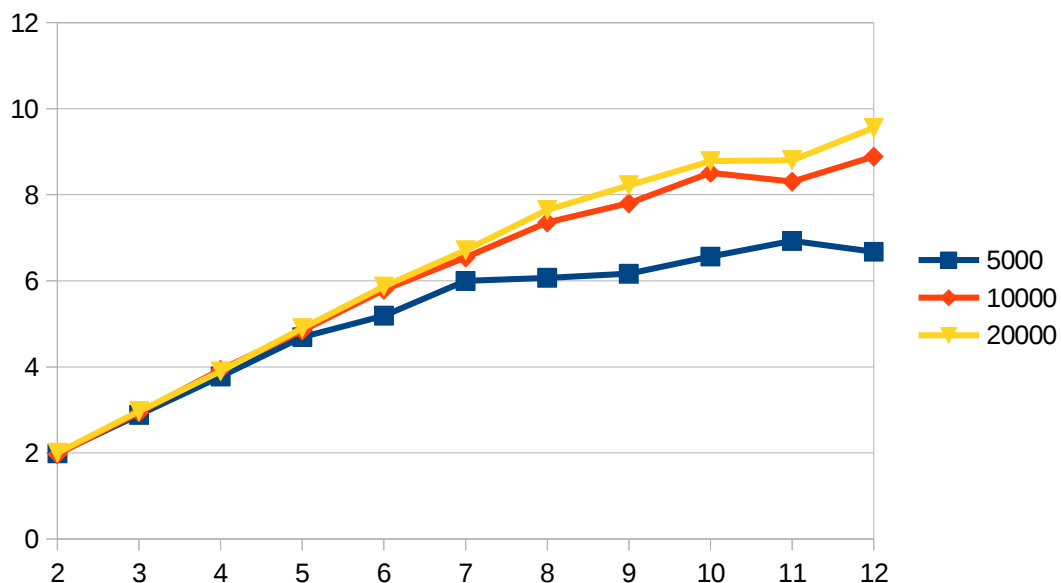
			A	5000	10000	20000
			2	0,0168649	0,0679769	0,2666276
			3	0,0116251	0,0455001	0,1797354
Seq	t		4	0,0088762	0,0339989	0,1369434
5000	0,0335234		5	0,007146	0,0276144	0,1090143
10000	0,1337202		6	0,0064621	0,0230941	0,0909439
20000	0,5337517		7	0,0055892	0,0204399	0,0795443
			8	0,0055224	0,0181816	0,0698027
			9	0,0054375	0,0171371	0,0649344
			10	0,005107	0,0157101	0,0607725
			11	0,0048392	0,0161004	0,0606332
			12	0,0050225	0,0150457	0,0558494

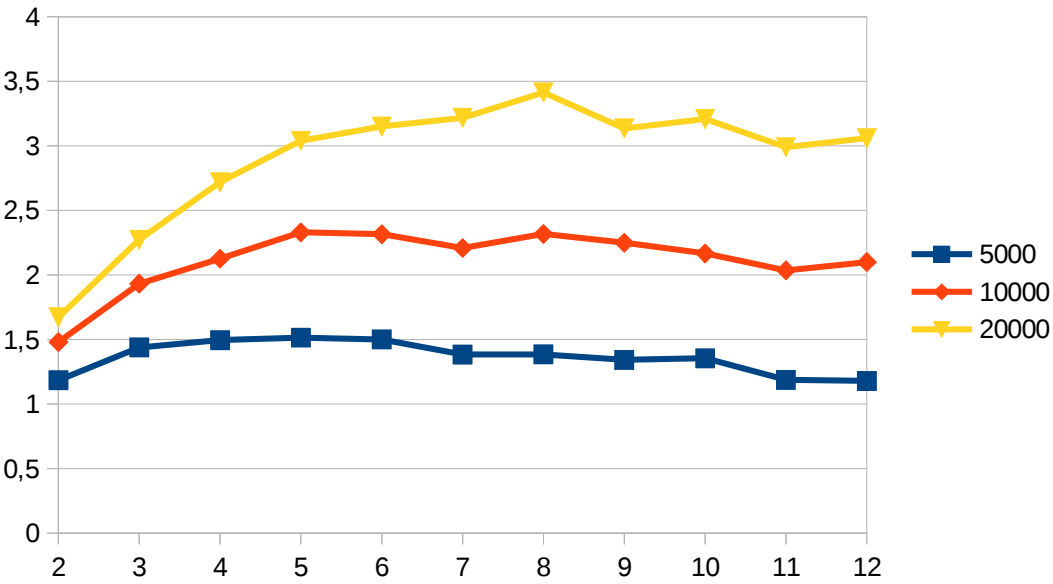
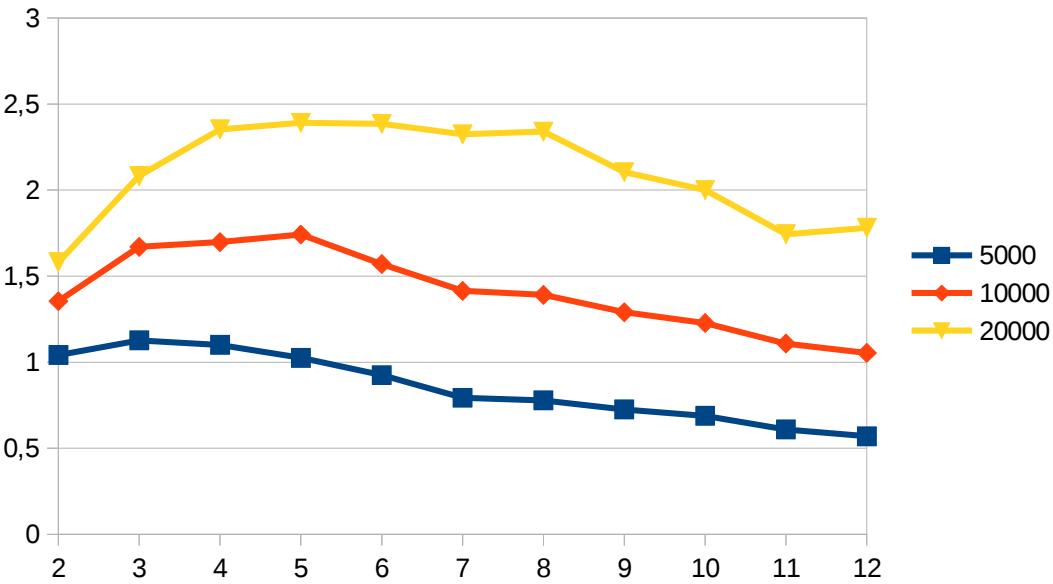
B	5000	10000	20000		C	5000	10000	20000
2	0,0321848	0,098732	0,3376226		2	0,0283139	0,0903854	0,3193539
3	0,029756	0,0800672	0,256332		3	0,0233011	0,0692157	0,2348699
4	0,0304626	0,0787608	0,2268427		4	0,0224306	0,0629003	0,1963622
5	0,0327108	0,0767898	0,223221		5	0,0221437	0,0574043	0,1755982
6	0,0362535	0,0851753	0,2238112		6	0,0223412	0,057757	0,1694124
7	0,0422764	0,0945794	0,2296608		7	0,0242479	0,0605412	0,1658615
8	0,0430919	0,0961126	0,2281145		8	0,0242163	0,0577101	0,1563445
9	0,0462188	0,1036496	0,2536532		9	0,0249872	0,0594594	0,1702658
10	0,0487134	0,1089155	0,2668233		10	0,0247513	0,0617028	0,1663413
11	0,0550701	0,1206718	0,3061055		11	0,0282435	0,0656967	0,17852
12	0,0589358	0,1268974	0,2996798		12	0,0284485	0,063682	0,1744842

Tabla general de ganancias:

A	5000	10000	20000	B	5000	10000	20000	C	5000	10000	20000
2	1,987761564	1,967141779	2,0018621478	2	1,0415910616	1,3543754811	1,5809122375	2	1,183990902	1,4794446891	1,6713486198
3	2,8837085272	2,9388990354	2,9696526116	3	1,1266097594	1,6700996163	2,0822671379	3	1,4387046105	1,9319345177	2,2725419477
4	3,7767738447	3,9330743053	3,8976080629	4	1,1004773066	1,6978014444	2,3529595618	4	1,4945387105	2,1259071896	2,7181998368
5	4,6912118668	4,8424083087	4,8961622466	5	1,0248419482	1,7413797145	2,3911356906	5	1,513902374	2,3294457035	3,0396194266
6	5,1876944027	5,7902321372	5,8690214517	6	0,9246941675	1,569941051	2,3848301604	6	1,5005192201	2,3152206659	3,1506058588
7	5,9978887855	6,5421161552	6,710118764	7	0,7929577731	1,4138406461	2,3240870884	7	1,3825279715	2,2087471011	3,2180566316
8	6,0704403882	7,3546992564	7,6465767084	8	0,7779513087	1,3912868864	2,3398411763	8	1,3843320408	2,3171022057	3,4139461254
9	6,1652229885	7,8029654959	8,2198603514	9	0,7253195669	1,2901178586	2,104257703	9	1,3416229109	2,2489328853	3,1348145077
10	6,5642059918	8,5117344893	8,7827833313	10	0,6881761487	1,2277426078	2,0003938936	10	1,3544096674	2,1671658336	3,2087743693
11	6,92746735	8,3053961392	8,802961084	11	0,6087404962	1,1081313115	1,7436854287	11	1,1869421283	2,0354173041	2,9898706027
12	6,6746441015	8,8876024379	9,5569818118	12	0,5688121651	1,053766271	1,7810733323	12	1,1783890188	2,0998115637	3,0590259748

Gráficas que representan la ganancia en función del número de threads utilizados (códigos A,B y C en orden):





COMENTARIOS SOBRE LOS RESULTADOS:

La primera reflexión que se me ocurre es que si vemos los datos con tranquilidad nos damos cuenta de una cosa: para cada tamaño n fijo cuando vamos aumentando el número de threads observamos que la ganancia queda estabilizada, tal y como predice la ley de Amdahl.

Sin embargo si leemos las tablas para tamaños sucesivos la ganancia no se ve limitada sino que crece casi linealmente tal y como predice la ley de Gustafson (toda aplicación de tamaño suficientemente grande puede ser paralelizada de manera eficiente).

Por otro lado, es claro que hay una dependencia con la arquitectura de la máquina. Si una arquitectura soporta más threads entonces obtendrá mejores ganancias.

Por último, es necesario hacer una comparación entre la eficiencia paralela de los códigos. Si tomamos para ello la ganancia como medida, se ve claramente que el código A es más eficiente que los códigos B y C. Para justificarlo hemos de observar que el código C utiliza la cláusula `reduction` de comunicación entre threads. En teoría hemos clasificado la comunicación entre threads como tiempo de sobrecarga y así es, el código C obtiene peores resultados que el A. Por su parte el código B acusa la presencia de la directiva `critical`. Evidentemente, el acceso exclusivo hace menos eficiente al código.

Se puede observar también que estos defectos se acentúan conforme aumenta el número de threads.