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):
Grupo de prácticas:
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: Con default(none) tenemos que especificar el ámbito de todas las variables que aparecen en la construcción, exceptuando aquellas variables de la cláusula threadprivate y los índices de en directivas con for. Para arreglar este problema, habría que añadir la variable n a la cláusula shared.

CAPTURA CÓDIGO FUENTE: `shared-clauseModificado.c`

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

void 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]);


}//fin del main
```

CAPTURAS DE PANTALLA:

```
guillesiesta@guillesiesta:~$ gcc shared-clauseModificado.c -o shMod -fopenmp
shared-clauseModificado.c: In function ‘main’:
shared-clauseModificado.c:15:10: error: ‘n’ not specified in enclosing parallel
  #pragma omp parallel for shared(a) default(none)
          ^
shared-clauseModificado.c:15:10: error: enclosing parallel
guillesiesta@guillesiesta:~$
```

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: Si se inicializa fuera del parallel la variable contendrá basura (un valor predeterminado), por lo que se ha de inicializar dentro de la sección del parallel o usar la cláusula firstprivate.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado.c`

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

void main(){

    int i, n=7;
    int a[n], suma;

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

    suma=1234567890;
    #pragma omp parallel private(suma)
    {
        suma=80;
        #pragma omp for
        for(i=0;i<n;i++){
            suma = suma + a[i];
            printf("\nthread %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:

```
guillesiesta@guillesiesta:~$ ./pr

thread 0 suma a[0] /
thread 0 suma a[1] /
thread 3 suma a[6] /
thread 2 suma a[4] /
thread 2 suma a[5] /
thread 1 suma a[2] /
thread 1 suma a[3] /
* thread 0 suma= 81
* thread 2 suma= 89
* thread 3 suma= 86
* thread 1 suma= 85
```

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

RESPUESTA: La variable suma pasa a ser compartida, por lo que todas las hebras podrán modificarla y machachar el valor previo.

CAPTURA CÓDIGO FUENTE: `private-clauseModificado3.c`

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

void 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("\nthread %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:

```
guillesiesta@guillesiesta:~$ ./prMod3

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 a[4] /
thread 2 suma a[5] /
* thread 0 suma= 15
* thread 1 suma= 15
* thread 3 suma= 15
* thread 2 suma= 15
guillesiesta@guillesiesta:~$ ./prMod3

thread 0 suma a[0] /
thread 0 suma a[1] /
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= 21
* thread 2 suma= 21
* thread 1 suma= 21
* thread 3 suma= 21
```

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: Lastprivate asigna a la variable el último valor que se asignaría en una ejecución secuencial, que en este caso siempre será 6 (0+6).

CAPTURAS DE PANTALLA:

```
guillesiesta@guillesiesta:~$ gcc firstprivate-clause.c -o fr -fopenmp
guillesiesta@guillesiesta:~$ ./fr

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
guillesiesta@guillesiesta:~$ ./fr

thread 0 suma a[0] suma=0
thread 0 suma a[1] suma=1
thread 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 3 suma a[6] suma=6
Fuera de la construcción parallel suma=6
```

5. ¿Qué se observa en los resultados de ejecución de `copyprivate-clause.c` cuando se elimina la cláusula `copyprivate(a)` en la directiva `single`? ¿A qué cree que es debido?

RESPUESTA: El funcionamiento es incorrecto. La variable "a" contiene basura en aquellos casos en los que está leída en alguna hebra, pero no ha sido copiada a las demás. Una vez termina single, ya sí se copia mediante difusión.

CAPTURA CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

```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 de a:");
            scanf("%d",&a);
            printf("\nSingle  ejectuada 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\n",i,b[i]);
    printf("\n");
}
```

CAPTURAS DE PANTALLA:

```
guillesiesta@guillesiesta:~$ gcc copyprivate-clauseModificado.c -o cp -fopenmp
guillesiesta@guillesiesta:~$ ./cp

Introduce valor de inicialización de a:5

Single  ejectuada por el thread 1
Después de la región parallel:
b[0] = 4196990
b[1] = 4196990
b[2] = 4196990
b[3] = 5
b[4] = 5
b[5] = 0
b[6] = 0
b[7] = 0
b[8] = 0
```

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

RESPUESTA: El programa suma todos los números desde 0 hasta el número que se pase por parámetro (tiene que ser menor que 20). Se van almacenando el resultado de las operaciones de la suma en la variable "suma", que inicialmente vale 0. Si le ponemos valor inicial 10, "suma" contendrá el resultado de la suma anteriormente descrita + 10.

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado.c`

```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:

```
guillesiesta@guillesiesta:~$ gcc reduction-clauseModificado.c -o rcM -fopenmp
guillesiesta@guillesiesta:~$ ./rcM 6
Tras 'parallel' suma=25
guillesiesta@guillesiesta:~$ ./rcM 5
Tras 'parallel' suma=20
guillesiesta@guillesiesta:~$ ./rcM 4
Tras 'parallel' suma=16
guillesiesta@guillesiesta:~$ ./rcM 3
Tras 'parallel' suma=13
guillesiesta@guillesiesta:~$ ./rcM 2
Tras 'parallel' suma=11
guillesiesta@guillesiesta:~$ ./rcM 1
Tras 'parallel' suma=10
guillesiesta@guillesiesta:~$
```

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: Hay que buscar la manera de repartir el trabajo para que cada hebra haga su parte. He usado el identificador de cada hebra y el número total de hebras, se ha implementado un round-robin. Es muy importante poner la cláusula private con la variable “i “.

CAPTURA CÓDIGO FUENTE: `reduction-clauseModificado7.c`

```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 private(i) reduction(+:suma)
    {
        for(i=omp_get_thread_num();i<n;i+=omp_get_num_threads()){
            suma += a[i];
            printf("\nthread %d suma a[%d] y suma vale: %d ", omp_get_thread_num(), i,suma);
        }
    }


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

CAPTURAS DE PANTALLA:

```
guillesiesta@guillesiesta:~$ gcc reduction-clauseModificado7.c -o rcM7 -fopenmp
guillesiesta@guillesiesta:~$ ./rcM7 5

thread 0 suma a[0] y suma vale: 0
thread 0 suma a[4] y suma vale: 4
thread 1 suma a[1] y suma vale: 1
thread 3 suma a[3] y suma vale: 3
thread 2 suma a[2] y suma vale: 2
Tras 'parallel' suma=10
guillesiesta@guillesiesta:~$
```

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).

Anotación: La versión hecha con variables globales está implementada en el archivo pmv-secuencialGlobales.c.

CAPTURA CÓDIGO FUENTE: `pmv-secuencial.c`

```c
// gcc -O2 algo.c -o nombre -fopenmp
#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** argv)
{
    int i, j;
    double t1, t2, total;

    //Argumento de entrada, N es el número de componentes del vector
    if (argc<2){
        printf("Falta tamaño de la matriz y del vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)

    double *v1, *v2, **M;
    v1 = (double*) malloc(N*sizeof(double));// malloc necesita el tamaño en bytes
    v2 = (double*) malloc(N*sizeof(double)); //si no hay espacio suficiente malloc devuelve NULL
    M = (double**) malloc(N*sizeof(double *));

    if ( (v1==NULL) || (v2==NULL) || (M==NULL) ){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }

    for (i=0; i<N; i++){
        M[i] = (double*) malloc(N*sizeof(double));
        if ( M[i]==NULL ){
            printf("Error en la reserva de espacio para la matriz\n");
            exit(-2);
        }
    }

    //Inicializar matriz y vectores
    for (i=0; i<N;i++){
        v1[i] = i;
        v2[i] = 0;
        for(j=0;j<N;j++)
            M[i][j] = i+j;
    }

    //Tomamos tiempo antes de operar
    t1 = omp_get_wtime();
```

```c
    //Calcular producto de matriz por vector M * v1 = v2
    for (i=0; i<N;i++)
        for(j=0;j<N;j++)
                v2[i] += M[i][j] * v1[j];

    //Tomamos tiempo después de operar
    t2 = omp_get_wtime();
    total = t2 - t1;

    //Imprimir el resultado primero y último, además del tiempo de ejecución
    printf("Tiempo(seg.):%11.9f\t / Tamaño:%u\t/ V2[0]=%8.6f V2[%d]=%8.6f\n", total,N,v2[0],N-1,v2
[N-1]);

    // Imprimir todos los componentes de v2 para valores de entrada no muy altos
    if (N<20){
        for (i=0; i<N;i++){
            printf(" V2[%d]=%5.2f\n", i, v2[i]);
        }
    }

    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
    for (i=0; i<N; i++)
        free(M[i]);
    free(M);

return 0;
}
```

CAPTURAS DE PANTALLA:

```
guillesiesta@guillesiesta:~$ gcc -O2 pmv-secuencial.c -o pmv-sec -fopenmp
guillesiesta@guillesiesta:~$ ./pmv-sec 11
Tiempo(seg.):0.000001394        / Tamaño:11    / V2[0]=385.000000 V2[10]=935.000000
 V2[0]=385.00
 V2[1]=440.00
 V2[2]=495.00
 V2[3]=550.00
 V2[4]=605.00
 V2[5]=660.00
 V2[6]=715.00
 V2[7]=770.00
 V2[8]=825.00
 V2[9]=880.00
 V2[10]=935.00
guillesiesta@guillesiesta:~$ ./pmv-sec 110
Tiempo(seg.):0.000076734        / Tamaño:110   / V2[0]=437635.000000 V2[109]=1091090.000000
guillesiesta@guillesiesta:~$
```

9. Implementar en paralelo el producto matriz por vector con OpenMP a partir del código escrito en el ejercicio anterior usando la directiva `for` . Debe implementar dos versiones del código (consulte la lección 5/Tema 2):

    a. una primera que paralelice el bucle que recorre las filas de la matriz y

    b. una segunda que paralelice el bucle que recorre las columnas.

    Use las directivas que estime oportunas y las cláusulas que sean necesarias **excepto la cláusula `reduction`**. Se debe paralelizar también la inicialización de las matrices. Respecto a este ejercicio:

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

    NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para tamaños pequeños de los vectores (por ejemplo, N = 8 y N=11); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

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

```c
// gcc -O2 algo.c -o nombre -fopenmp
#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** argv)
{
    int i, j;
    double t1, t2, total;

    //Argumento de entrada, N es el número de componentes del vector
    if (argc<2){
        printf("Falta tamaño de la matriz y del vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]);

    double *v1, *v2, **M;
    v1 = (double*) malloc(N*sizeof(double));
    v2 = (double*) malloc(N*sizeof(double));
    M = (double**) malloc(N*sizeof(double *));

    if ( (v1==NULL) || (v2==NULL) || (M==NULL) ){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }

    for (i=0; i<N; i++){
        M[i] = (double*) malloc(N*sizeof(double));
        if ( M[i]==NULL ){
            printf("Error en la reserva de espacio para la matriz\n");
            exit(-2);
        }
    }

    //Inicializar matriz y vectores
    for (i=0; i<N;i++){
        v1[i] = i;
        v2[i] = 0;
        for(j=0;j<N;j++)
            M[i][j] = i+j;
    }

    //Tomamos tiempo antes de operar
    t1 = omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp for
        //Calcular producto de matriz por vector M * v1 = v2
        for (i=0; i<N;i++){
            for(j=0;j<N;j++){
                    v2[i] += M[i][j] * v1[j];
            }
        }
    }
    //Tomamos tiempo después de operar
    t2 = omp_get_wtime();
    total = t2 - t1;

    //Imprimir el resultado primero y último, además del tiempo de ejecución
    printf("Tiempo(seg.):%11.9f\t / Tamaño:%u\t/ V2[0]=%8.6f V2[%d]=%8.6f\n", total,N,v2[0],N-1,v2
[N-1]);

    // Imprimir todos los componentes de v2 para valores de entrada no muy altos
    if (N<20){
        for (i=0; i<N;i++){
            printf(" V2[%d]=%5.2f\n", i, v2[i]);
        }
    }

    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
    for (i=0; i<N; i++)
        free(M[i]);
    free(M);

return 0;
}
```

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

```c
// gcc -O2 algo.c -o nombre -fopenmp
#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** argv)
{
    int i, j;
    double t1, t2, total;

    //Argumento de entrada, N es el número de componentes del vector
    if (argc<2){
        printf("Falta tamaño de la matriz y del vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)

    double *v1, *v2, **M;
    v1 = (double*) malloc(N*sizeof(double));// malloc necesita el tamaño en bytes
    v2 = (double*) malloc(N*sizeof(double)); //si no hay espacio suficiente malloc devuelve NULL
    M = (double**) malloc(N*sizeof(double *));

    if ( (v1==NULL) || (v2==NULL) || (M==NULL) ){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }

    for (i=0; i<N; i++){
        M[i] = (double*) malloc(N*sizeof(double));
        if ( M[i]==NULL ){
            printf("Error en la reserva de espacio para la matriz\n");
            exit(-2);
        }
    }

    //Inicializar matriz y vectores
    for (i=0; i<N;i++){
        v1[i] = i;
        v2[i] = 0;
        for(j=0;j<N;j++)
            M[i][j] = i+j;
    }

    //Tomamos tiempo antes de operar
    t1 = omp_get_wtime();


```

```
54
55          //Calcular producto de matriz por vector M * v1 = v2
56          for (i=0; i<N;i++){
57              #pragma omp parallel
58              {
59                  double acumula = 0;
60                  #pragma omp for
61                  for(j=0;j<N;j++){
62                      acumula += M[i][j] * v1[j];
63                  }
64                  #pragma omp critical
65                  v2[i] +=acumula;
66              }
67
68          }
69
70      //Tomamos tiempo después de operar
71      t2 = omp_get_wtime();
72      total = t2 - t1;
73
74      //Imprimir el resultado primero y último, además del tiempo de ejecución
75      printf("Tiempo(seg.):%11.9f\t / Tamaño:%u\t/ V2[0]=%8.6f V2[%d]=%8.6f\n", total,N,v2[0],N-1,v2
    [N-1]);
76
77      // Imprimir todos los componentes de v2 para valores de entrada no muy altos
78      if (N<20){
79          for (i=0; i<N;i++){
80              printf(" V2[%d]=%5.2f\n", i, v2[i]);
81          }
82      }
83
84      free(v1); // libera el espacio reservado para v1
85      free(v2); // libera el espacio reservado para v2
86      for (i=0; i<N; i++)
87          free(M[i]);
88      free(M);
89
90  return 0;
91  }
```

RESPUESTA: En la primera versión no he tenido ningún problema de compilación y los resultados estaban igual que en la versión secuencial. Sin embargo, a la hora de implementar la segunda versión, los resultados salían distintos. Esto se debe que a la hora de paralelizar la operación y hacerlo por columnas, se está escribiendo en el vector v2 sin seguir ningún orden, por lo que los resultados son distintos, algunas hebras acceden antes de que otras. Por lo que he decidido acumular cada resultado de una hebra en una variable temporal llamada “acumula” y después crear una sección crítica a la hora de almacenar ese valor en el lugar que le corresponde en el vector resultante v2. Pero me encuentro otro problema y el programa no se ejecuta, se queda atascado. Pero si me salto el protocolo que dijo el profesor y creo las hebras, es decir, añado la línea #pragma omp parallel justo debajo del for que recorre las filas, el programa continúa con su ejecución y los resultados son correctos.

CAPTURAS DE PANTALLA:

```
guillesiesta@guillesiesta:~$ gcc -O2 pmv-OpenMP-a.c -o pmv-OpenMP-a -fopenmp
guillesiesta@guillesiesta:~$ ./pmv-OpenMP-a 20
Tiempo(seg.):0.003721005         / Tamaño:20    / V2[0]=2470.000000 V2[19]=6080.000000
guillesiesta@guillesiesta:~$ ./pmv-sec 20
Tiempo(seg.):0.000003093         / Tamaño:20    / V2[0]=2470.000000 V2[19]=6080.000000
guillesiesta@guillesiesta:~$
```

```
Tiempo(seg.):0.000003093         / Tamaño:20    / V2[0]=2470.000000 V2[19]=6080.000000
guillesiesta@guillesiesta:~$ gcc -O2 pmv-OpenMP-b.c -o pmv-OpenMP-b -fopenmp
guillesiesta@guillesiesta:~$ ./pmv-OpenMP-b 20
Tiempo(seg.):0.007452188         / Tamaño:20    / V2[0]=2470.000000 V2[19]=6080.000000
guillesiesta@guillesiesta:~$
```

10. A partir de la segunda versión de código paralelo desarrollado en el ejercicio anterior, implementar una versión paralela del producto matriz por vector con OpenMP que use para comunicación/sincronización la cláusula `reduction`. Respecto a este ejercicio:
    - Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
    - Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

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

```c
// gcc -O2 algo.c -o nombre -fopenmp
#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** argv)
{
    int i, j;
    double t1, t2, total;

    //Argumento de entrada, N es el número de componentes del vector
    if (argc<2){
        printf("Falta tamaño de la matriz y del vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295 (sizeof(unsigned int) = 4 B)

    double *v1, *v2, **M;
    v1 = (double*) malloc(N*sizeof(double));// malloc necesita el tamaño en bytes
    v2 = (double*) malloc(N*sizeof(double)); //si no hay espacio suficiente malloc devuelve NULL
    M = (double**) malloc(N*sizeof(double *));

    if ( (v1==NULL) || (v2==NULL) || (M==NULL) ){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }

    for (i=0; i<N; i++){
        M[i] = (double*) malloc(N*sizeof(double));
        if ( M[i]==NULL ){
            printf("Error en la reserva de espacio para la matriz\n");
            exit(-2);
        }
    }

    //Inicializar matriz y vectores
    for (i=0; i<N;i++){
        v1[i] = i;
        v2[i] = 0;
        for(j=0;j<N;j++)
            M[i][j] = i+j;
    }

    //Tomamos tiempo antes de operar
    t1 = omp_get_wtime();
```

```
54
55          //Calcular producto de matriz por vector M * v1 = v2
56
57
58          for (i=0; i<N;i++){
59                  double acumula = 0;
60                  #pragma omp parallel
61                  {
62                      #pragma omp for reduction(+:acumula)
63                      for(j=0;j<N;j++){
64                          acumula += M[i][j] * v1[j];
65                      }
66                      v2[i] =acumula;
67                  }
68          }
69
70      //Tomamos tiempo después de operar
71      t2 = omp_get_wtime();
72      total = t2 - t1;
73
74      //Imprimir el resultado primero y último, además del tiempo de ejecución
75      printf("Tiempo(seg.):%11.9f\t / Tamaño:%u\t/ V2[0]=%8.6f V2[%d]=%8.6f\n", total,N,v2[0],N-1,v2
  [N-1]);
76
77      // Imprimir todos los componentes de v2 para valores de entrada no muy altos
78      if (N<20){
79          for (i=0; i<N;i++){
80              printf(" V2[%d]=%5.2f\n", i, v2[i]);
81          }
82      }
83
84      free(v1); // libera el espacio reservado para v1
85      free(v2); // libera el espacio reservado para v2
86      for (i=0; i<N; i++)
87          free(M[i]);
88      free(M);
89
90  return 0;
91  }
```

RESPUESTA: De esta manera se simplifica bastante el código. Sin embargo, OpenMP no permite escribir el resultado de un reduction en una posición de un vector, por lo que he tenido que guardar el resultado en la variable “acumula” y después asignárselo al vector.

CAPTURAS DE PANTALLA:

```
guillesiesta@guillesiesta:~$ gcc -O2 pmv-OpenMP-reduction.c -o pmv-red -fopenmp
guillesiesta@guillesiesta:~$ ./pmv-red 20
Tiempo(seg.):0.004318192         / Tamaño:20    / V2[0]=2470.000000 V2[19]=6080.000000
guillesiesta@guillesiesta:~$ ./pmv-sec 20
Tiempo(seg.):0.000003174         / Tamaño:20    / V2[0]=2470.000000 V2[19]=6080.000000
guillesiesta@guillesiesta:~$
```

11. Ayudándose de una hoja de cálculo (recuerde que en las aulas está instalado OpenOffice) realice una tabla y una gráfica que permitan comparar la escalabilidad (ganancia en velocidad en función del número de cores) en atcgrid y en su PC del mejor código paralelo de los tres implementados en los ejercicios anteriores para dos tamaños (N) distintos (consulte la Lección 6/Tema 2). Usar –O2 al compilar. Justificar por qué el código escogido es el mejor. NOTA: Nunca ejecute en atcgrid código que imprima todos los componentes del resultado.

CAPTURAS DE PANTALLA (que justifique el código elegido):

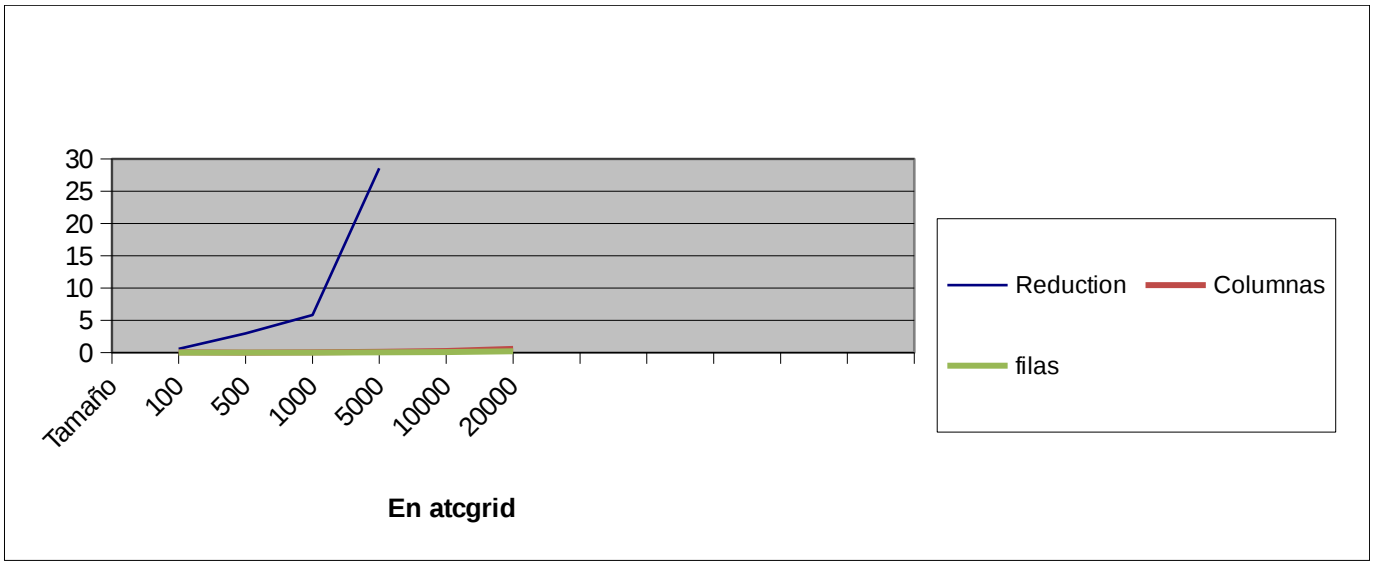
No hace falta captura, en la tabla se puede comprobar rápidamente que el código más rápido es el realizado para paralelizar por filas.

TABLA Y GRÁFICA (por *ejemplo* para 1-4 threads PC local, y para 1-12 threads en atcgrid, tamaños-N-: un N entre 30000 y 100000, y otro entre 5000 y 30000):

Aclaración: Mi PC solo puede tirar como máximo con 20000. Entonces he decidido comparar con el mismo valor en ATCGRID para que sea más sencillo observar la diferencia de tiempos para una misma entrada N.

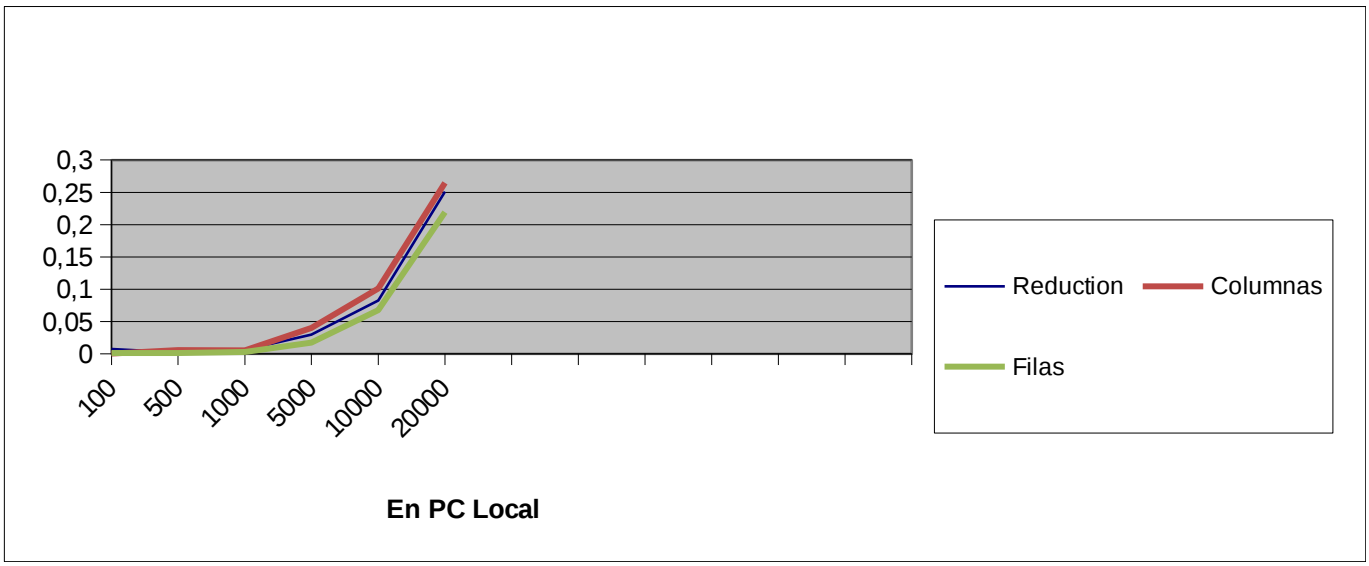
ATCGRID

Tamaño	Filas	Columnas	Reduction
100	0.006768025	0.037906934	0.571600770
500	0.004378495	0.011244566	2.953310519
1000	0.005286004	0.024748721	5.819291417
5000	0.016119187	0.097577314	28.549429218
10000	0.056095919	0.254964762	+60 seg
20000	0.211767059	0.573936435	+60 seg



PC-LOCAL

Tamaño	Filas	Columnas	Reduction
100	0.001142197	0.000438393	0.007374358
500	0.001279621	0.005541032	0.002181745
1000	0.002978287	0.005087072	0.005901898
5000	0.017356539	0.040179353	0.029787523
10000	0.067533303	0.100974370	0.082422942
20000	0.219223813	0.264653856	0.251101149



COMENTARIOS SOBRE LOS RESULTADOS:

Los resultados son muy llamativos. De primeras, se observa perfectamente que el mejor código tanto en PC como en ATCGRID es la primera versión implementada, en la que se paraleliza por filas.

Por último, no me explico porque en atcgrid me han salido tiempos muchísimo peores que en mi PC local (intel i7). Supongo que esto tiene que ser debido a que atcgrid se encontraba realizando otro tipo de operaciones de gran capacidad computacional al mismo tiempo que yo estaba tomando mis medidas.

Llama mucho la atención que la peor versión sea la que usa la cláusula reduction. Se simplifica mucho el código, pero no es eficiente.