

Arquitectura de Computadores (AC)

Cuaderno de prácticas.

Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Néstor Rodríguez Vico

Grupo de prácticas: A1

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: Da error a la hora de compilar.

CÓDIGO FUENTE: `shared-clauseModificado.c`

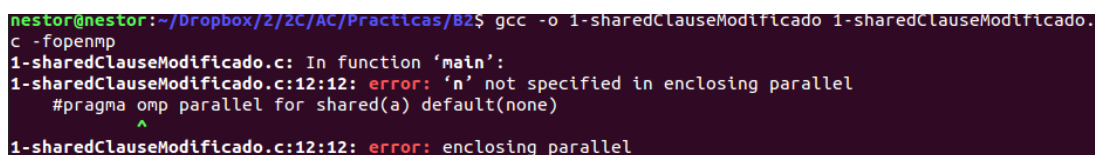
```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
int main(int argc, char **argv) {
    int i, n = 7;
    int a[n];

    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:



```
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/BZ$ gcc -o 1-sharedClauseModificado 1-sharedClauseModificado.c -fopenmp
1-sharedClauseModificado.c: In function 'main':
1-sharedClauseModificado.c:12:12: error: 'n' not specified in enclosing parallel
    #pragma omp parallel for shared(a) default(none)
                       ^
1-sharedClauseModificado.c:12:12: error: enclosing parallel
```

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

RESPUESTA: Si inicializamos la variable `suma` fuera de `parallel` no estará inicializada dentro, ya que `private` lo único que hace es crear para cada hebra una copia pero no le da un valor. Si usamos, por ejemplo, `-O2` al compilar, ningún valor 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 argc, char **argv) {
    int i, n = 7;
    int a[n], suma;

    for (i=0; i<n; i++)
        a[i] = i;
    suma = 0;
    #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* Fuera de la region Parallel: thread %d suma= %d",
    omp_get_thread_num(), suma);
    printf("\n");
}

```

CAPTURAS DE PANTALLA:

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/82$ ./2-private-clauseModificado
thread 2 suma a[4] / thread 2 suma a[5] / 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 1 suma= 4196613
* thread 0 suma= 5
* thread 2 suma= 4196617
* thread 3 suma= 4196614
* Fuera de la region Parallel: thread 0 suma= 0

```

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

RESPUESTA: El resultado es el mismo ya que todas las hebras comparten la misma variable “suma” y el resultado que se muestra es el que tuviese la ultima hebra en salir del bucle.

CÓDIGO FUENTE: `private-clauseModificado3.c`

```

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

int main(int argc, char **argv) {
    int i, n = 7;
    int a[n], suma;

    for (i=0; i<n; i++)
        a[i] = i;
    suma = 0;
    #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: thread %d suma= %d",
omp_get_thread_num(), suma);
    printf("\n");
}

```

CAPTURAS DE PANTALLA:

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./3-private-clauseModificado3
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= 17
* thread 0 suma= 17
* thread 3 suma= 17
* thread 2 suma= 17
* Fuera de la region Parallel: thread 0 suma= 17

```

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

CAPTURAS DE PANTALLA:

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./4-firstlastprivate-clause
thread 2 suma a[4] suma=4
thread 2 suma a[5] suma=9
thread 0 suma a[0] suma=0
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
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./4-firstlastprivate-clause
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 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
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./4-firstlastprivate-clause
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 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

```

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

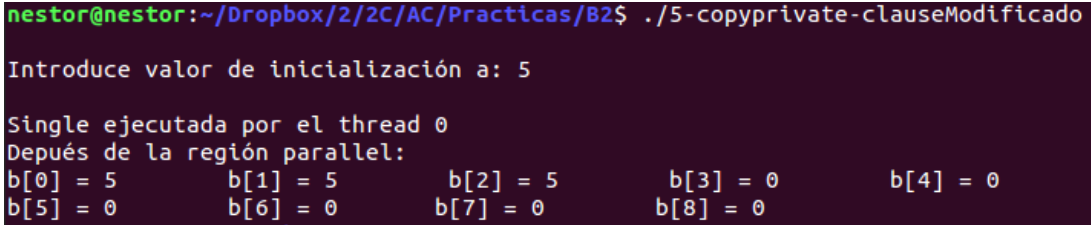
RESPUESTA: `copyprivate(a)` difunde el valor de “a” a las otras hebras. Al eliminar `copyprivate(a)` no se produce la difusión y únicamente se inicializan correctamente aquellas zonas que haya inicializado la hebra que ejecutó el `single`.

CÓDIGO FUENTE: copyprivate-clauseModificado.c

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

int main(int argc, char **argv) {
    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:


```
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./5-copyprivate-clauseModificado
Introduce valor de inicialización a: 5

Single ejecutada por el thread 0
Después de la región parallel:
b[0] = 5      b[1] = 5      b[2] = 5      b[3] = 0      b[4] = 0
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: Sale el mismo resultado que con suma = 0 pero sumándole 10. Se debe a que reduction mantiene el valor inicial de la variable de acumulación (suma).

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:

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./6-reduction-clauseModificado 2
Tras 'parallel' suma=11
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./6-reduction-clauseModificado 10
Tras 'parallel' suma=55
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./6-reduction-clauseModificado 15
Tras 'parallel' suma=115

```

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 sin usar directivas de trabajo compartido .

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=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
{
    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:

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./7-reduction-clauseModificado7 2
Tras 'parallel' suma=1
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./7-reduction-clauseModificado7 5
Tras 'parallel' suma=10
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B2$ ./7-reduction-clauseModificado7 10
Tras 'parallel' suma=45

```

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):

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

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

```

// Compilar con -O2 y -fopenmp
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i, j;
    double t1, t2, total, res;

    //Leer argumento de entrada (no de componentes del vector)
    if (argc<2){
        printf("Falta tamaño de matriz y 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 los vectores\n");
            exit(-2);
        }
    }

    //A partir de aqui se pueden acceder las componentes de la
matriz como M[i][j]

    //Inicializar matriz y vectores

    for(i=0; i<N; i++) {
        v1[i]=1;
    }

    for(i=0; i<N; i++) {
        for(j=0; j<N; j++) {
            M[i][j]=1;
        }
    }

    //Medida de tiempo
    t1 = omp_get_wtime();

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

    //Medida de tiempo
    t2 = omp_get_wtime();
    total = t2 - t1;

    //Imprimir el resultado y el 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]);

    if (N < 30){
        for(i=0; i<N; i++)
            printf("%f ", v2[i]);
        printf("\n");
    }

    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:

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/5. Bloques/B2$ ./8-pmv-secuencial_02 8
Tiempo(seg.): 0.000000225 / Tamaño:8 / V2[0]=8.000000 V2[7]=8.000000
8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/5. Bloques/B2$ ./8-pmv-secuencial_02 11
Tiempo(seg.): 0.000000411 / Tamaño:11 / V2[0]=11.000000 V2[10]=11.000000
11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000
11.000000

```

9. Implementar en paralelo el producto matriz por vector con OpenMP a partir del código escrito en el ejercicio anterior usando la directiva `for`. Debe implementar dos versiones del código (consulte la lección 5/Tema 2):
 - a. una primera que paralelice el bucle que recorre las filas de la matriz y
 - b. una segunda que paralelice el bucle que recorre las columnas.

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

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

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

CÓDIGO FUENTE : pmv-OpenMP-a.c

```

// Compilar con -O2 y -fopenmp
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i, j;
    double t1, t2, total;

    //Leer argumento de entrada (no de componentes del vector)
    if (argc<2){
        printf("Falta tamaño de matriz y 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 los vectores\n");
                exit(-2);
            }
        }

        //A partir de aqui se pueden acceder las componentes de la
matriz como M[i][j]

#pragma omp parallel
{
    //Inicializar matriz y vectores

    #pragma omp for
    for(i=0; i<N; i++)
        v1[i]=1;

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

    //Medida de tiempo
    #pragma omp single
    {
        t1 = omp_get_wtime();
    }

    //Calcular producto de matriz por vector v2 = M .
v1

    #pragma omp parallel for private(j)
    for(i=0; i<N; i++) {
        v2[i]=0;
        for(j=0; j<N; j++) {
            v2[i]+=(M[i][j]*v1[j]);
        }
    }

    //Medida de tiempo
    #pragma omp single
    {
        t2 = omp_get_wtime();
    }

}

total = t2 - t1;

//Imprimir el resultado y el 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]);

```

```

    if (N < 30){
        for(i=0; i<N; i++)
            printf("%f ", v2[i]);
        printf("\n");
    }

    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;
}

```

CÓDIGO FUENTE: pmv-OpenMP-b.c

```

// Compilar con -O2 y -fopenmp
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i, j;
    double t1, t2, total;

    //Leer argumento de entrada (no de componentes del vector)
    if (argc<2){
        printf("Falta tamaño de matriz y 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 los vectores\n");
            exit(-2);
        }
    }

    //A partir de aqui se pueden acceder las componentes de la
matriz como M[i][j]

    #pragma omp parallel private(i)
    {
        //Inicializar matriz y vectores

```

```

#pragma omp for
for(i=0; i<N; i++)
    v1[i]=1;

#pragma omp for
for(i=0; i<N; i++) {
    for(j=0; j<N; j++) {
        M[i][j]=1;
    }
}

//Medida de tiempo
#pragma omp single
{
    t1 = omp_get_wtime();
}

//Calcular producto de matriz por vector v2 = M .
v1
    for(i=0; i<N; i++) {
        v2[i]=0;
        double res=0;
        #pragma omp for
        for(j=0; j<N; j++) {
            res+=(M[i][j]*v1[j]);
        }
        #pragma omp atomic
        v2[i]+=res;
    }

    //Medida de tiempo
    #pragma omp single
    {
        t2 = omp_get_wtime();
    }
}

total = t2 - t1;

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

    i=0, j=0;
    if (N < 30){
        for(i=0; i<N; i++)
            printf("%f ", v2[i]);
        printf("\n");
    }

    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;
}

```

RESPUESTA: La mayoría de errores han sido a la hora de compilar, por lo general a la hora de decidir si las variables deben ser compartidas o no, es decir, al aplicar la regla

general que se vio en el seminario.

Otro error común ha sido a la hora de manejar las variables que controlan el for. Este error ha sido en ejecución, al ver que los resultados obtenidos no eran los correctos.

Para solucionarlo, he tenido que revisar las variables una por una para ver si su ámbito era el correcto. Tras revisarlas y ajustar los ámbitos de las variables empleadas en mi programa, la solución obtenida ha sido la correcta.

CAPTURAS DE PANTALLA:

```
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/5. Bloques/B2$ ./9-pmv-OpenMP-a_02 8
Tiempo(seg.): 0.000001842 / Tamaño:8 / V2[0]=8.000000 V2[7]=8.000000
8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/5. Bloques/B2$ ./9-pmv-OpenMP-a_02 11
Tiempo(seg.): 0.000770041 / Tamaño:11 / V2[0]=11.000000 V2[10]=11.000000
11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000
```

```
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/5. Bloques/B2$ ./9-pmv-OpenMP-b_02 8
Tiempo(seg.): 0.000061570 / Tamaño:8 / V2[0]=8.000000 V2[7]=8.000000
8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/5. Bloques/B2$ ./9-pmv-OpenMP-b_02 11
Tiempo(seg.): 0.00018245 / Tamaño:11 / V2[0]=11.000000 V2[10]=11.000000
11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000
```

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-OpenMP-reduction.c

```
// Compilar con -O2 y -fopenmp
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
    int i, j;
    double t1, t2, total, res=0;

    //Leer argumento de entrada (no de componentes del vector)
    if (argc<2){
        printf("Falta tamaño de matriz y 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 los vectores\n");
                                exit(-2);
            }
        }

        //A partir de aqui se pueden acceder las componentes de la
matriz como M[i][j]

#pragma omp parallel private(i)
{
    //Inicializar matriz y vectores
#pragma omp for
    for(i=0; i<N; i++)
        v1[i]=1;

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

    //Medida de tiempo
#pragma omp single
    {
        t1 = omp_get_wtime();
    }

    //Calcular producto de matriz por vector v2 = M .
v1
    for(i=0; i<N; i++) {
        #pragma omp for reduction(+:res)
        for(j=0; j<N; j++) {
            res+=(M[i][j]*v1[j]);
        }
        #pragma omp single
        {
            v2[i]=res;
            res = 0;
        }
    }

    //Medida de tiempo
#pragma omp single
    {
        t2 = omp_get_wtime();
    }

}

total = t2 - t1;

//Imprimir el resultado y el 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]);

```

```

    if (N < 30){
        for(i=0; i<N; i++){
            printf("%lf ", v2[i]);
        }
        printf("\n");
    }

    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;
}

```

RESPUESTA: Mas o menos los mismos que en el ejercicio anterior.

CAPTURAS DE PANTALLA:

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/5. Bloques/B2$ ./10-pmv-OpenMMP-reduction_02 8
Tiempo(seg.): 0.000009265 / Tamaño:8 / V2[0]=8.000000 V2[7]=8.000000
8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000 8.000000
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/5. Bloques/B2$ ./10-pmv-OpenMMP-reduction_02 11
Tiempo(seg.): 0.000022749 / Tamaño:11 / V2[0]=11.000000 V2[10]=11.000000
11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000 11.000000
11.000000

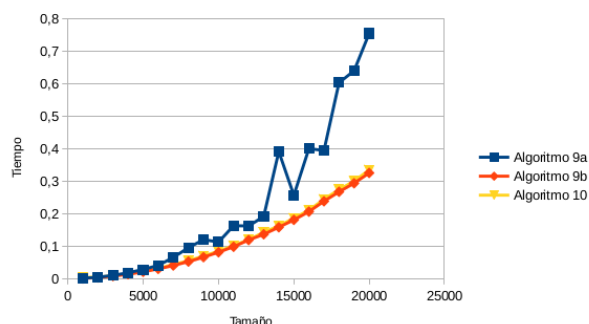
```

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.

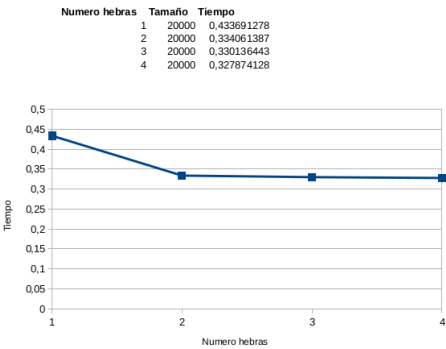
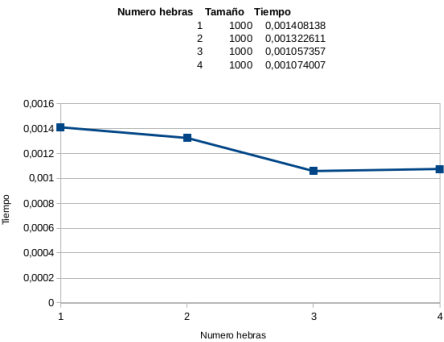
TABLA Y GRÁFICA (por ejemplo para 1-4 threads PC local, y para 1-12 threads en atcgrid, tamaños-N: alguno del orden de cientos de miles):

En mi caso, el mejor algoritmo es el 9b, podemos verlo en la siguiente imagen:

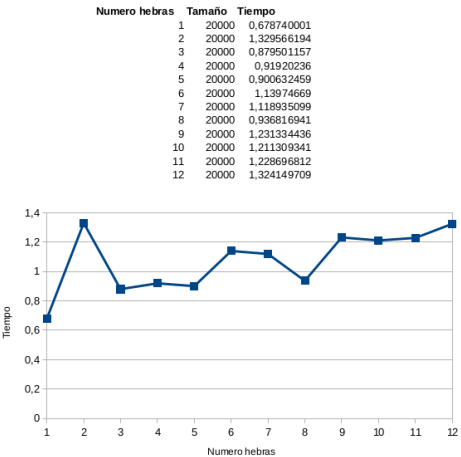
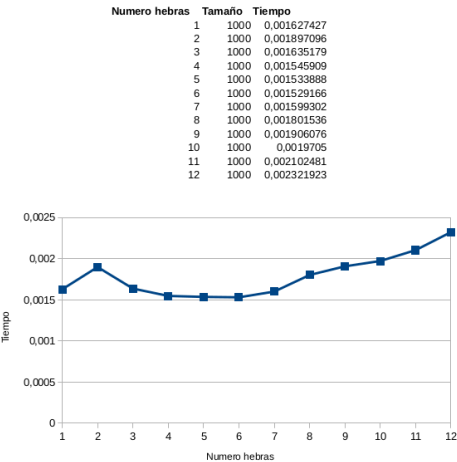
Tamaño	Algoritmo 9a	Algoritmo 9b	Algoritmo 10
1000	0,002133747	0,001055555	0,004356427
2000	0,005000454	0,004121146	0,004739787
3000	0,010837272	0,008405344	0,009705407
4000	0,018639271	0,01415288	0,015336079
5000	0,028788499	0,021574689	0,023207181
6000	0,041218487	0,030639035	0,032368472
7000	0,065705028	0,040913262	0,043746474
8000	0,096247305	0,052994123	0,055894825
9000	0,12048285	0,066604325	0,070344005
10000	0,11329905	0,081831008	0,084833711
11000	0,163767632	0,098843195	0,102563508
12000	0,162862684	0,119489865	0,12060761
13000	0,192101304	0,137176935	0,143446726
14000	0,391940481	0,158833228	0,162813627
15000	0,257238168	0,181516195	0,1866724
16000	0,400916504	0,207015067	0,21132764
17000	0,395175784	0,238911402	0,244978444
18000	0,604096984	0,2680353	0,274662285
19000	0,640726987	0,294332051	0,301120795
20000	0,75579606	0,326285029	0,335253636



Por lo tanto el estudio lo voy a hacer sobre dicho algoritmo.
Los resultados obtenidos para el pc del aula son:



Los resultados obtenidos para el atcgrid son:



COMENTARIOS SOBRE LOS RESULTADOS:

Podemos ver que los tiempos no siempre mejoran conforme aumenta el numero de hebras.