Cuaderno de prácticas
de Arquitectura de Computadores
Grado en Ingeniería Informática

Memoria
Bloque Práctico 2

Alumno: María Oliver Balsalobre
DNI: 75719191-V
Grupo: Viernes 10-12 h

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)`.

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 default(none) shared(a) firstprivate(n)

  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:

En primer lugar, adjunto una captura de pantalla de la ejecución del programa `shared-clause` sin modificaciones.

```
marycachi@marycachiPC:~$ ./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
marycachi@marycachiPC:~$
```

Si le añadimos `default(none)` tal y como muestra la siguiente imagen, nos da un error de compilación, que también muestro a continuación:

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

```
marycachi@marycachiPC:~$ gcc -O2 -fopenmp -o shared-clauseModificado shared-clauseModificado.c
shared-clauseModificado.c: In function ‘main’:
shared-clauseModificado.c:14:11: error: ‘n’ not specified in enclosing parallel
shared-clauseModificado.c:14:11: error: enclosing parallel
marycachi@marycachiPC:~$
```

Este error nos indica que la variable `n` no está especificada dentro de la directiva `parallel`. Lo solucionaremos añadiendo al final `firsprivate(n)` tal y como se muestra en el código `shared-clauseModificado.c`.

Con esta directiva, forzamos a que la variable `n` tenga el valor que tenía anteriormente, y no que empiece a contar desde 0, como pasaría si pusiéramos la directiva `private(n)` en lugar de `firsprivate(n)`.

A continuación vemos que con esta orden, `firstprivate(n)`, el error está resuelto, sin necesidad de eliminar `default(none)`.

```
marycachi@marycachiPC:~$ gcc -O2 -fopenmp -o shared-clauseModificado shared-clau
seModificado.c
marycachi@marycachiPC:~$ ./shared-clauseModificado
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
marycachi@marycachiPC:~$
```

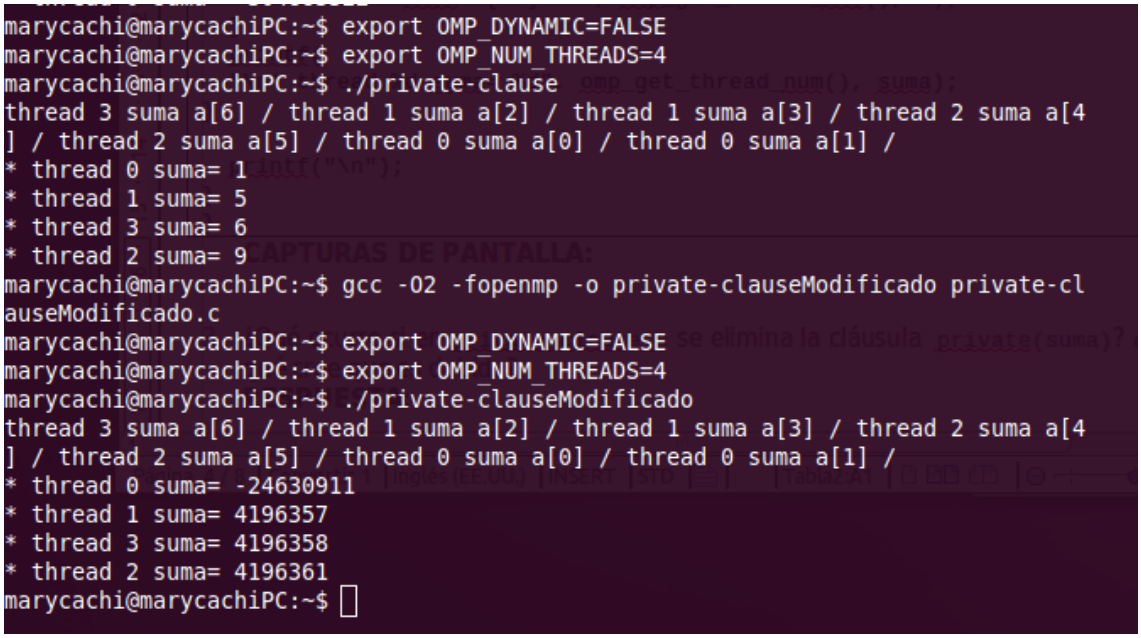
2. ¿Qué ocurre si en `private-clause.c` se inicializa la variable `suma` fuera de la construcción `parallel` en lugar de dentro? Razone su respuesta.

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;

  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");

}
```

CAPTURAS DE PANTALLA:

Veremos la ejecución con el programa sin modificar, y posteriormente la ejecución del programa modificado tal y como se indica en el enunciado.



RESPUESTA:

Como vemos, el valor de la suma en cada thread no tiene nada que ver en un programa y en el otro. Esto es porque al declarar la variable suma fuera de la construcción `parallel`, como estamos empleando la directiva `private`, el valor de entrada y salida está indefinido aunque la variable esté declarada fuera de la construcción, es decir, los valores de entrada no están declarados al entrar en la directiva, y no se guardará su valor de salida.

Cada hebra tendrá su propia variable suma.

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

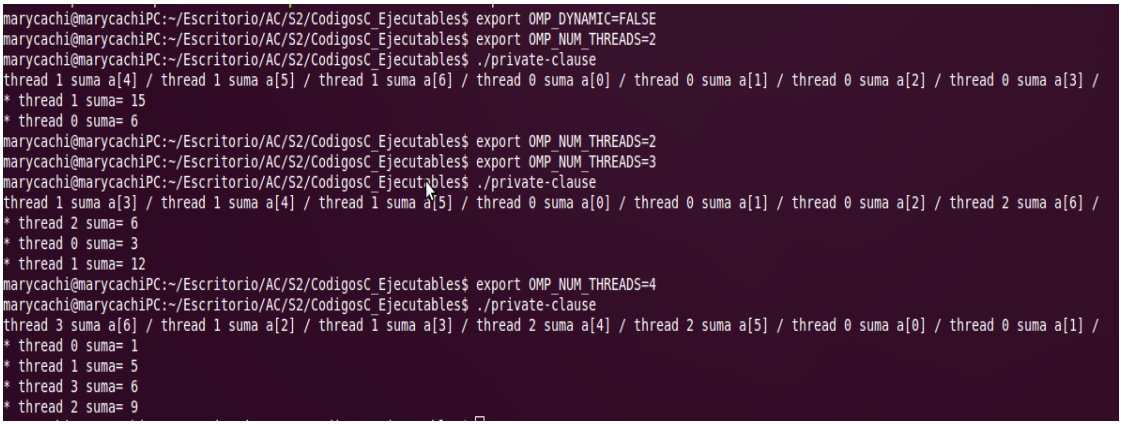
CÓDIGO FUENTE: `private-clauseModificado3.c`

```
#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");

}
```

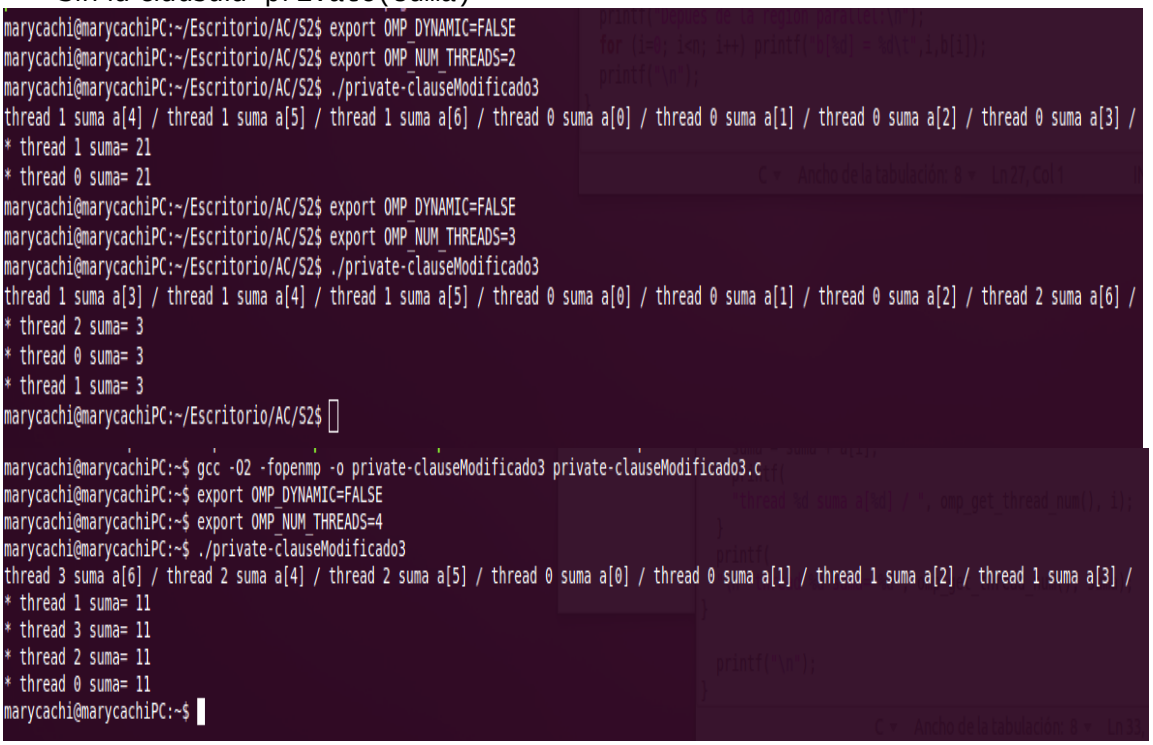
CAPTURAS DE PANTALLA:

Lo vemos para 2, 3 y 4 threads.

Sin modificar:



Sin la cláusula `private(suma)`



RESPUESTA:

Como vemos, ahora el resultado de la variable `suma` es el mismo para todos los threads. Esto es debido a que con la directiva `private`, cada hebra tenía su propio valor de esta variable `suma`. Al quitar dicha directiva, todas las hebras comparten el mismo valor de la variable y por eso el programa muestra el mismo resultado en todos los hilos.

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:

Como podemos ver en las capturas de pantalla, efectivamente el código siempre imprime suma = 6 cuando lo ejecutamos.
Para las variables definidas con la cláusula `lastprivate`, se hace una copia privada de dicha variable en cada hebra pero con la particularidad de que al final del bloque paralelo el valor de dicha variable se mantiene al valor que toma en la última iteración.
En este ejercicio, cuando acaba el bloque paralelo, el valor de la variable es siempre 6, y de ahí que siempre muestre el mismo resultado sea cual sea el número de hebras.

CAPTURAS DE PANTALLA:

```
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ export OMP_DYNAMI
C=FALSE
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ export OMP_NUM_TH
READS=3
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ ./firstlastprivat
e
 thread 0 suma a[0] suma=0
 thread 0 suma a[1] suma=1
 thread 0 suma a[2] suma=3
 thread 2 suma a[6] suma=6
 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=6
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ export OMP_NUM_TH
READS=4
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ ./firstlastprivat
e
 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
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ export OMP_NUM_TH
READS=5
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ ./firstlastprivat
e
 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
 thread 3 suma a[6] suma=6
```

```
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ export OMP_NUM_TH
READS=6
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ ./firstlastprivat
e
 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
 thread 2 suma a[4] suma=4
 thread 2 suma a[5] suma=9
```

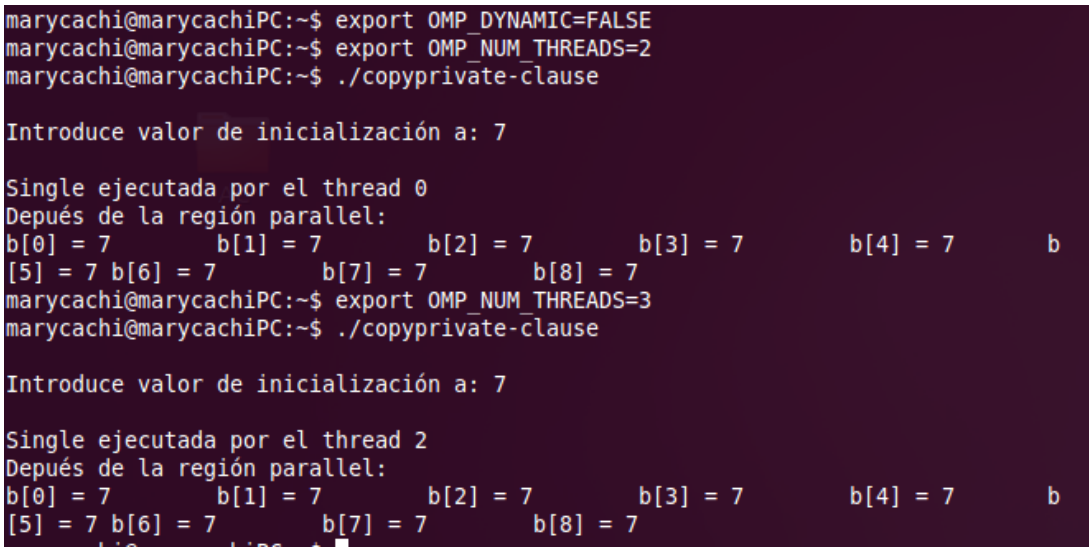
5. ¿Qué ocurre si en `copyprivate-clause.c` se elimina la cláusula `copyprivate(a)` en la directiva `single`? ¿A qué cree que es debido?

CÓDIGO FUENTE: `copyprivate-clauseModificado.c`

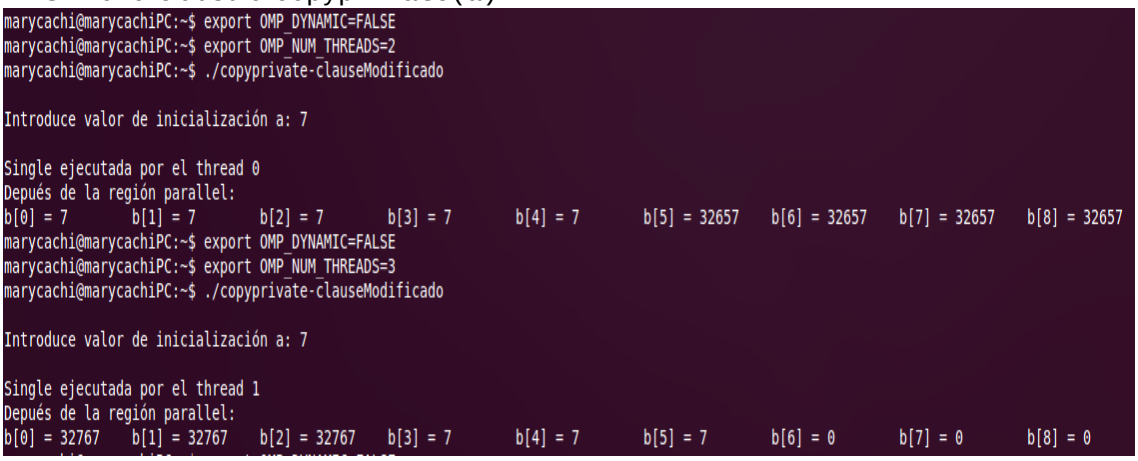
```
#include <stdio.h>
#include <omp.h>

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("Depués de la región parallel:\n");

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

CAPTURAS DE PANTALLA:

Sin modificar



Sin la la cláusula `copyprivate(a)`



RESPUESTA:

La directiva single sirve para que un sólo hilo ejecute un trozo de bloque, es decir, la parte de código que define esta directiva sólo puede ejecutarla una única hebra de todas las lanzadas, y no tiene que ser obligatoriamente la principal. Al eliminar la cláusula `copyprivate(a)`, la ejecución del programa cambia, tal y como se muestra en las capturas de pantalla.

El valor que le pongo en la inicialización sólo me lo muestra para una hebra, (en el caso de número de hilos = 2), y como he puesto 2 hebras, lo muestra para la mitad de los valores.

En el caso del número de hilos = 3, el valor de la inicialización sólo se muestra para un tercio de los valores (en la captura de pantalla, como es ejecutada por el thread 1, lo muestra para los valores 3,4 y 5).

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

CÓDIGO FUENTE: `reduction-clauseModificado.c`

```
#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);

}
```

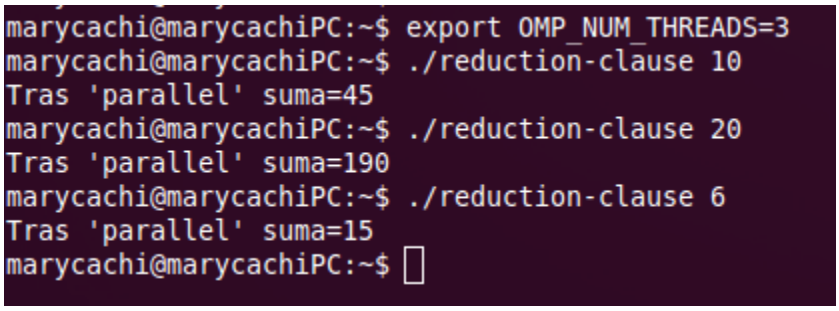
RESPUESTA:

Como podemos ver, al modificar el valor de inicialización de la variable `suma`, los resultados de la ejecución son distintos.
Concretamente, el resultado final de `suma`, tras las modificación, es +10. Esto es debido a que en el total de la suma también se incluye el valor de inicialización que tenía dicha variable. Como al principio `suma = 0`, esto no se apreciaba, pero al cambiar el valor de inicio a `suma = 10`, se puede ver con claridad que éste también se incluye en el cálculo final.

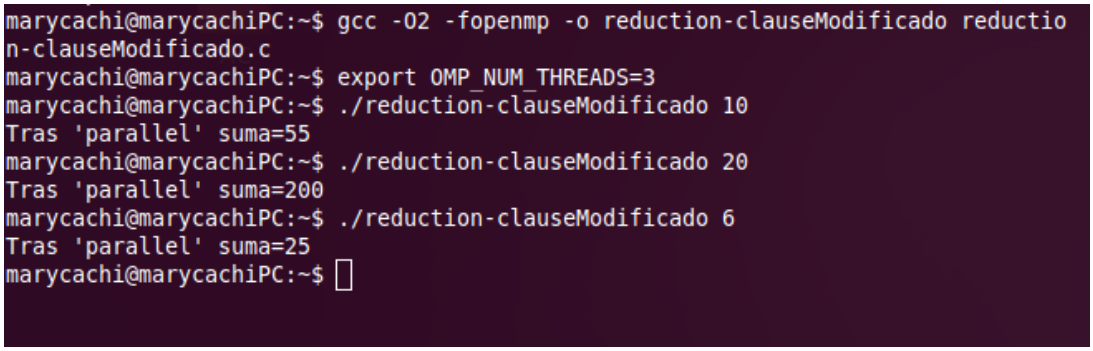
(Si inicializáramos `suma = 6`, tras 'parallel', al ejecutar la orden `./reduction-clauseModificado 10`, por ejemplo, suma=51).

CAPTURAS DE PANTALLA:

Sin modificar



Sustituyendo `suma=0` por `suma=10`



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.

CÓDIGO FUENTE: `reduction-clauseModificado7.c`

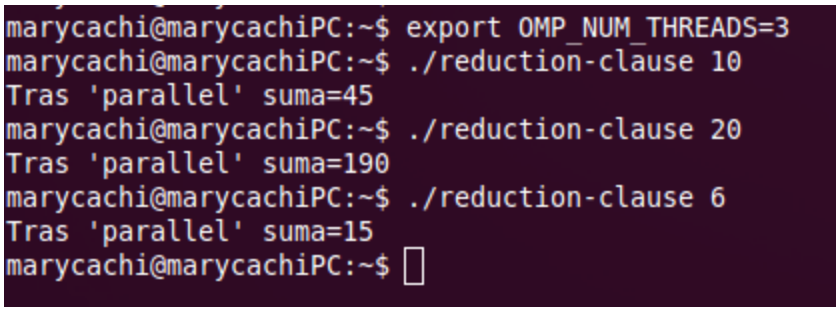
```
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

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);}

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

  #pragma omp barrier
  #pragma omp parallel for

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

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

}
```

RESPUESTA:

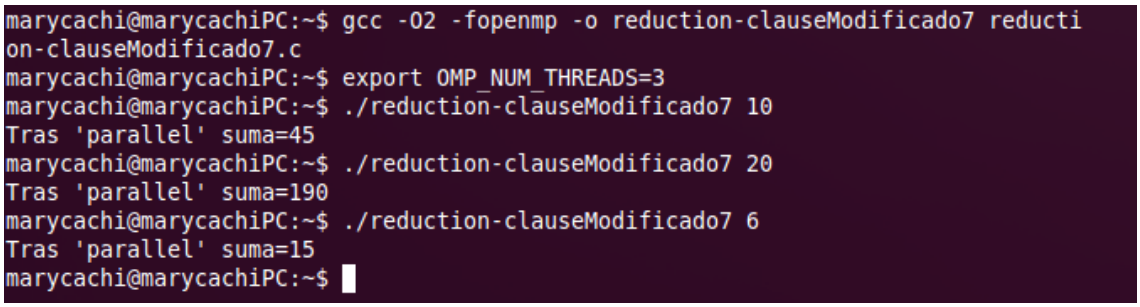
Al eliminar `reduction` de `#pragma omp parallel for reduction(+:suma)`, para que se siga realizando la suma de los componentes del vector `a` en paralelo, basta con añadir la directiva barrier, que, como vimos en el anterior Seminario, especifica una barrera en la que todos y cada uno de los threads deberá esperar al otro y, una vez que todos hayan llegado a la barrera, continuarán su trabajo en paralelo.

CAPTURAS DE PANTALLA:

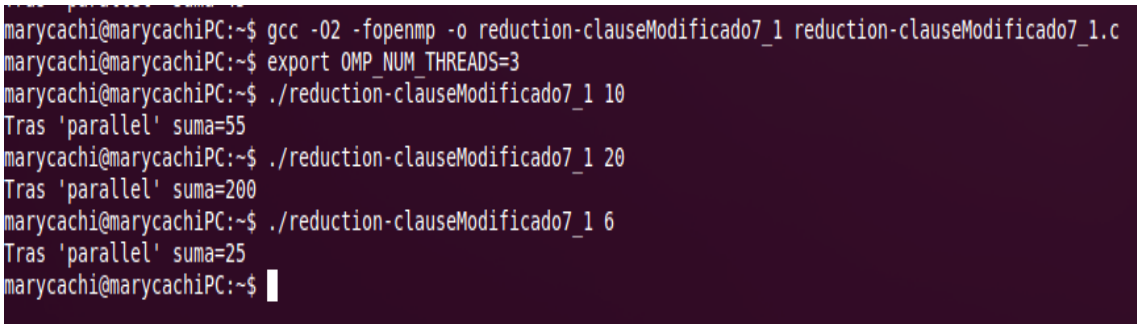
Sin modificar



Eliminando `reduction` de `#pragma omp parallel for reduction(+:suma)`, con la directiva `barrier`, y con `suma = 0`.



Eliminando `reduction` de `#pragma omp parallel for reduction(+:suma)`, con la directiva `barrier`, y con `suma = 10`.



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

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

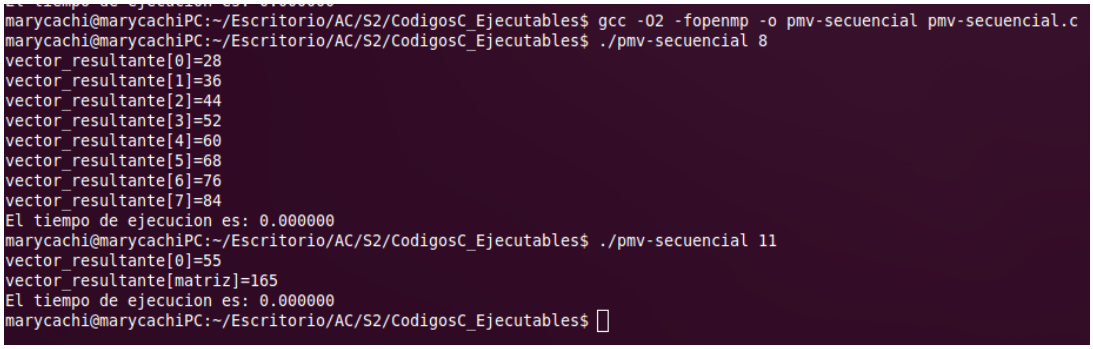
NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada al programa; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para tamaños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

CÓDIGO FUENTE: `pmv-secuencial.c`

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


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

  int i, j, k;
  int *vector;
  int **matriz;
  int *vector_resultante;

  if (argc < 2){
            fprintf(stderr, "\nError --> Falta el número de
componentes\n");
            return(-1);
  }

  int m;

  m = atoi(argv[1]);

  if (m > 11000){
            printf("Error --> extensión inadecuada\n");
            return(-1);
  }

  vector = (int *)malloc(m * sizeof(int));
  vector_resultante = (int *)malloc(m * sizeof(int));
  matriz = (int **)malloc(m * sizeof(int *));

  if (matriz == NULL){
            printf("Error de memoria \n");
            free(matriz);
            return -1;
  }
```

```
  for (i = 0;i < m;i++){
            matriz[i] = (int *)malloc(m * sizeof(int));
  }

  for (j = 0; j < m; j++){
            for (i = 0;i < m;i++){
            matriz[j][i] = j+i;
            }
            vector[j] = 1;
  }

  int suma;
  clock_t tiempo_antes, tiempo_despues;


  suma = 0;
  tiempo_antes = clock();


  for (i = 0; i < m; i++){
            suma=0;
            for (j = 0; j < m; j++){
                        suma+=(matriz[j][i]*vector[i]);
            }
            vector_resultante[i] = suma;
  }


  tiempo_despues = clock();

  if (m < 10){
            for (k = 0; k < m; k++){
                        printf ("vector_resultante[%d]=%d\n", k,
vector_resultante[k]);
            }
  }
  else{
            printf ("vector_resultante[0]=%d\n",vector_resultante[0]);
            printf ("vector_resultante[matriz]=
%d\n",vector_resultante[m-1]);

  }

  printf ("El tiempo de ejecución es: %f\n", (double)(tiempo_despues-
tiempo_antes),CLOCKS_PER_SEC);


}
```

CAPTURAS DE PANTALLA:



Para N = 8 imprimimos todos los componentes del vector resultante y para N = 11 solamente el primer y el último componente del resultado. En ambos se imprime el tiempo de ejecución del código paralelo que calcula el producto matriz por vector.

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):

    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).

    CÓDIGO FUENTE : `pmv-OpenMP-a.c`

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

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

  int k, j, i;
  double tiempo;

  int *vector;
  int **matriz;
  int *vector_resultante;

  if (argc < 2){
            fprintf(stderr, "\nError --> Falta el número de
componentes\n");
            return(-1);
  }
  int m;

  m = atoi(argv[1]);

  if (m > 11000){
            printf("Error --> extensión inadecuada\n");
            return(-1);
  }
  vector = (int *)malloc(m * sizeof(int));
  vector_resultante = (int *)malloc(m * sizeof(int));
  matriz = (int **)malloc(m * sizeof(int *));
```

```
  if (matriz == NULL){
            printf("Error de memoria \n");
            free(matriz);
            return -1;
  }

  for (i = 0;i < m;i++){
            matriz[i] = (int *)malloc(m * sizeof(int));

            if (matriz[i] == NULL){
                        printf("Error de memoria \n");
                        return -1;
            }
  }

  #pragma omp parallel for private(i)

  for (j = 0; j < m; j++){
            for (i = 0;i < m;i++){
            matriz[j][i] = j+i;
            }
            vector[j] = 1;
  }
  int suma;

  suma = 0;
  tiempo = omp_get_wtime();

  #pragma omp parallel for private(suma)
  for (i = 0; i < m; i++){
            suma = 0;

            for (j = 0; j < m; j++){
                        suma += (matriz[j][i]*vector[i]);
            }
            vector_resultante[i] = suma;
  }
  tiempo = omp_get_wtime() - tiempo;

  if (m < 10){
            for (k = 0; k < m; k++){
                        printf ("vector_resultante[%d]=%d\n", k,
vector_resultante[k]);
            }
  }
  else{
            printf ("vector_resultante[0]=%d\n",vector_resultante[0]);
            printf ("vector_resultante[matriz]=
%d\n",vector_resultante[m-1]);
  }

  printf ("El tiempo de ejecucion es: %f\n", tiempo);

}
```

CÓDIGO FUENTE: pmv-OpenMP-b.c

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

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

  int k, j, i;
  double tiempo;

  int *vector;
  int **matriz;
  int *vector_resultante;

  if (argc < 2){
            fprintf(stderr, "\nError --> Falta el número de
componentes\n");
            return(-1);
  }

  int m;


  m = atoi(argv[1]);

  if (m > 11000){
            printf("Error--> extension inadecuada\n");
            return(-1);
  }

  vector = (int *)malloc(m * sizeof(int));
  vector_resultante = (int *)malloc(m * sizeof(int));
  matriz = (int **)malloc(m * sizeof(int *));

  if (matriz == NULL){
            printf("Error de memoria \n");
            free(matriz);
            return -1;
  }

  for (i = 0;i < m;i++){
            matriz[i] = (int *)malloc(m * sizeof(int));
            if (matriz[i] == NULL){
                printf("Error de memoria \n");
                return -1;
            }
  }
```

```
  #pragma omp parallel for private(i)
  for (j = 0; j < m; j++){
            for (i = 0;i < m;i++){
            matriz[j][i] = j+i;
            }
            vector[j] = 1;
  }

  int suma, suma_local;

  suma = 0;
  tiempo = omp_get_wtime();

  for (i = 0; i < m; i++){
            suma = 0;

            #pragma omp parallel for private(suma_local)
            for (k = 0; k < m; k++){
                        suma_local = 0;
                        suma += (matriz[k][i]*vector[i]);
                        #pragma omp atomic
                        suma += suma_local;
            }
            vector_resultante[i] = suma;
  }

  tiempo = omp_get_wtime() - tiempo;

  if (m < 10){
            for (k = 0; k < m; k++){
                        printf ("vector_resultante[%d]=%d\n", k,
vector_resultante[k]);
            }
  }
  else{
            printf ("vector_resultante[0]=%d\n",vector_resultante[0]);
            printf ("vector_resultante[matriz]=
%d\n",vector_resultante[m-1]);
  }

  printf ("El tiempo de ejecución es: %f\n", tiempo);


}
```

RESPUESTA:

El código lo hemos hecho, en gran parte, en grupo en las horas de clase dedicadas para la realización de esta práctica. Con respecto al apartado a) (paralelizar las filas), no hemos tenido problemas demasiado graves porque en este caso, todas las hebras accedían a todas las componentes de la matriz pero no las modificaban.
El problema lo hemos tenido al paralelizar las columnas (apartado b)), ya que en este caso, las hebras, además de acceder a las componentes de la matriz, también las modifican.
Este inconveniente lo hemos resuelto con una variable declarada como privada, `suma_local`, para que cada hebra pueda tener la suya propia. Posteriormente esta nueva variable sería la suma total que muestra el resultado. Esto lo hemos hecho con ayuda de la directiva `atomic`, debido a que ésta se utiliza cuando la operación accede sólo a una posición de memoria y tiene que ser actualizada sólo por un hilo (además de que la operación de tipo += utiliza esta cláusula).

CAPTURAS DE PANTALLA:

a. Paralelizando el bucle que recorre las filas de la matriz

```
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ gcc -O2 -fopenmp -o pmv-OpenMP-a pmv-OpenMP-a.c
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ ./pmv-OpenMP-a 8
vector_resultante[0]=28
vector_resultante[1]=36
vector_resultante[2]=44
vector_resultante[3]=52
vector_resultante[4]=60
vector_resultante[5]=68
vector_resultante[6]=76
vector_resultante[7]=84
El tiempo de ejecución es: 0.000050
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ ./pmv-OpenMP-a 11
vector_resultante[0]=55
vector_resultante[matriz]=165
El tiempo de ejecución es: 0.000051
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$
```

b. Paralelizando el bucle que recorre las columnas.

```
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ gcc -O2 -fopenmp -o pmv-OpenMP-b pmv-OpenMP-b.c
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ ./pmv-OpenMP-b 8
vector_resultante[0]=28
vector_resultante[1]=36
vector_resultante[2]=44
vector_resultante[3]=52
vector_resultante[4]=60
vector_resultante[5]=68
vector_resultante[6]=76
vector_resultante[7]=84
El tiempo de ejecución es: 0.000394
marycachi@marycachiPC:~/Escritorio/AC/S2/CodigosC_Ejecutables$ ./pmv-OpenMP-b 11
vector_resultante[0]=55
vector_resultante[matriz]=165
El tiempo de ejecución es: 0.000490
```

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

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

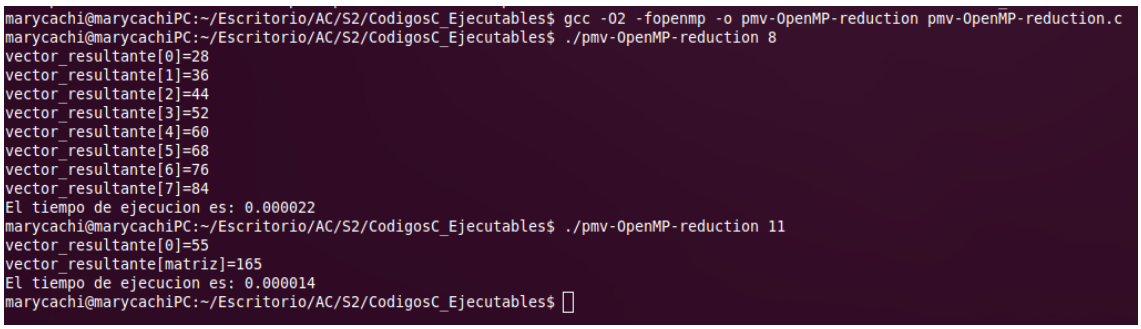
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#ifdef _OPENMP
#include <omp.h>
#endif

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


  int k, j, i;
  double tiempo;

  int *vector;
  int **matriz;
  int *vector_resultante;

  if (argc < 2){
            fprintf(stderr, "\nError --> Falta el número de
componentes\n");
            return(-1);
  }

  int m;

  m = atoi(argv[1]);

  if (m > 11000){
            printf("Error --> extensión inadecuada\n");
            return(-1);
  }

  vector = (int *)malloc(m * sizeof(int));
  vector_resultante = (int *)malloc(m * sizeof(int));
  matriz = (int **)malloc(m * sizeof(int *));

  if (matriz == NULL){
            printf("Error de memoria \n");
```

```
            free(matriz);
            return -1;
  }

  for (i = 0;i < m;i++){
            matriz[i] = (int *)malloc(m * sizeof(int));

            if (matriz[i] == NULL){
                        printf("Error de memoria \n");
                        return -1;
            }
  }

  #pragma omp parallel for private(i)
  for (j = 0; j < m; j++){
            for (i = 0;i < m;i++){
            matriz[j][i] = j+i;
            }
            vector[j] = 1;
  }

  int suma;

  suma = 0;
  tiempo = omp_get_wtime();

  for (i = 0; i < m; i++){
            suma = 0;

            #pragma omp parallel for reduction(+:suma)
            for (j = 0; j < m; j++){
                        suma += (matriz[j][i]*vector[i]);
            }
            vector_resultante[i] = suma;
  }

  tiempo = omp_get_wtime() - tiempo;

  if (m < 10){
            for (k = 0; k < m; k++){
                        printf ("vector_resultante[%d]=%d\n", k,
vector_resultante[k]);
            }
  }
  else{
            printf ("vector_resultante[0]=%d\n",vector_resultante[0]);
            printf ("vector_resultante[matriz]=
%d\n",vector_resultante[m-1]);
  }
  printf ("El tiempo de ejecución es: %f\n", tiempo);


}
```

RESPUESTA:

Este ejercicio, como era a partir del apartado b del ejercicio anterior, no ha ocasionado grandes dificultades tampoco.
Ahora, con la cláusula `reduction` hemos podido resolver el problema directamente, sin necesidad de utilizar la variable `suma_local` como privada, ya que permite especificar una o más variables privadas de subprocesos que están sujetas a una operación de reducción al final de la región paralela.
Las ayudas externas han sido internet, la ayuda del profesor y el trabajo en grupo para la elaboración del código.

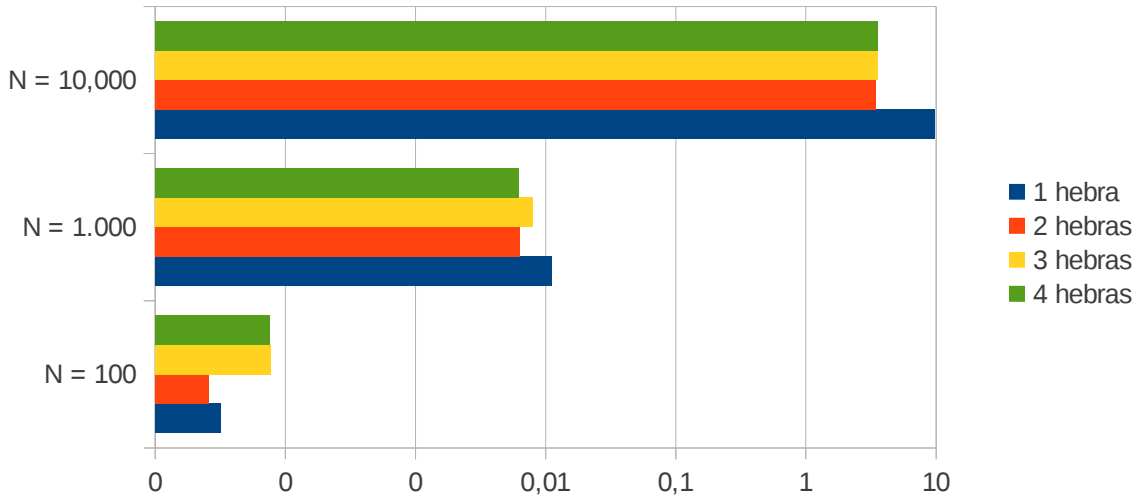
CAPTURAS DE PANTALLA:



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).

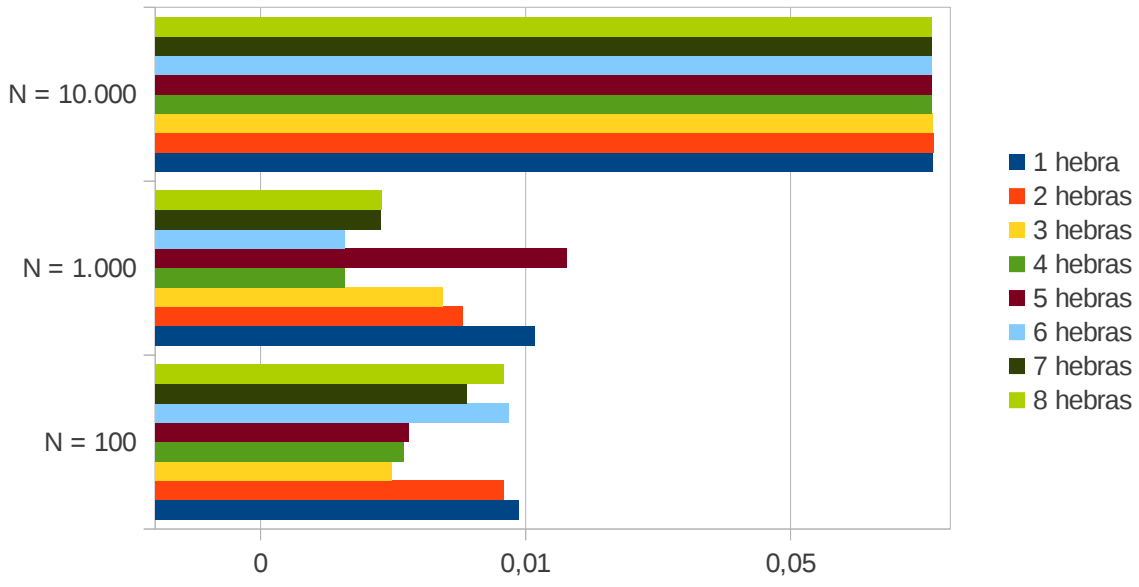
PC-local : pmv-OpenMP-a *(paralelización por filas)*

	N = 100	**N = 1.000**	**N = 10.000**
1 hebra	0.000032	0.011127	9.779030
2 hebras	0.000026	0.006377	3.464428
3 hebras	0.000077	0.008020	3.566690
4 hebras	0.000076	0.006212	3.542942



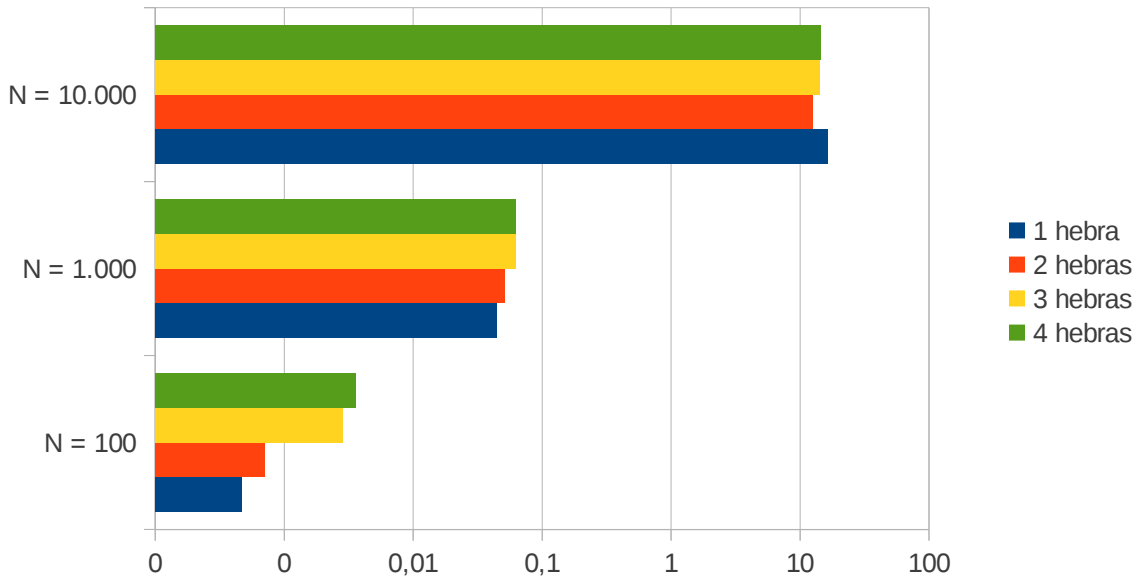
atcgrid : pmv-OpenMP-a *(paralelización por filas)*

	N = 100	N = 1.000	N = 10.000
1 hebra	0.004698	0.005430	0.172034
2 hebras	0.004117	0.002902	0.173012
3 hebras	0.001555	0.002431	0.172266
4 hebras	0.001736	0.001042	0.170616
5 hebras	0.001805	0.007170	0.170154
6 hebras	0.004317	0.001041	0.170099
7 hebras	0.002995	0.001420	0.170108
8 hebras	0.004149	0.001431	0.170439



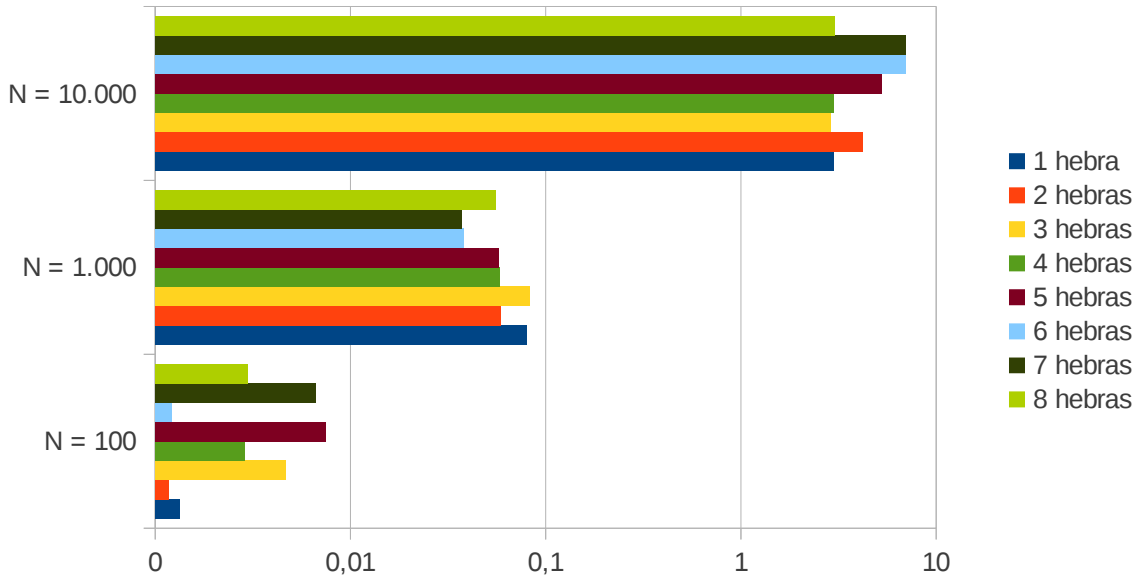
PC-local : pmv-OpenMP-b *(paralelización por columnas)*

	N = 100	N = 1.000	N = 10.000
1 hebra	0.000471	0.044653	16.386610
2 hebras	0.000712	0.051450	12.597939
3 hebras	0.002854	0.062197	14.173440
4 hebras	0.003562	0.062790	14.414016



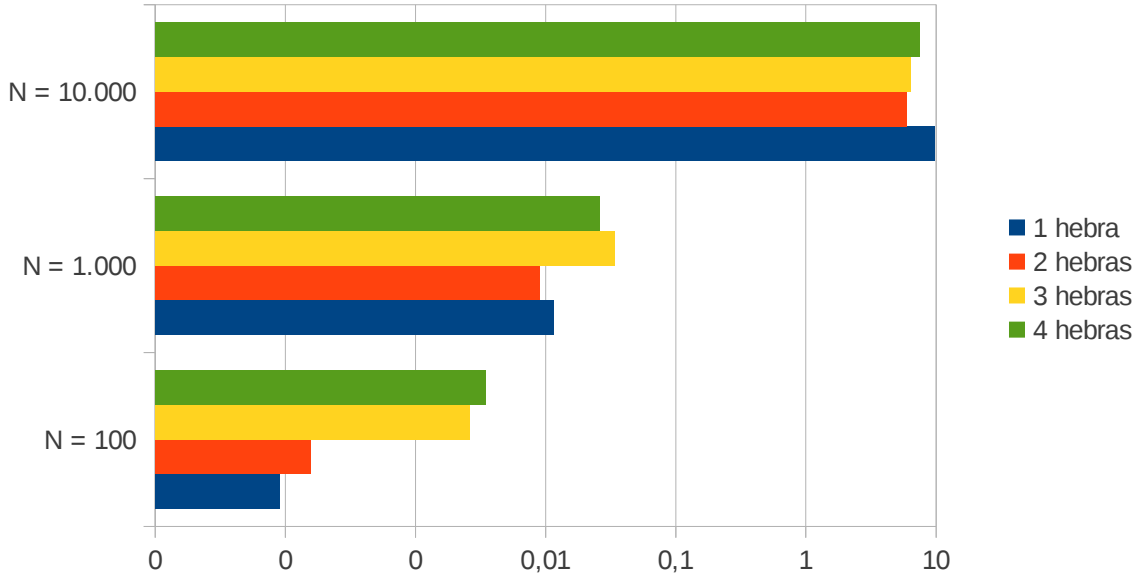
atcgrid : pmv-OpenMP-b *(paralelización por columnas)*

	N = 100	N = 1.000	N = 10.000
1 hebra	0.001330	0.080031	2.993125
2 hebras	0.001177	0.058994	4.199455
3 hebras	0.004637	0.082553	2.901313
4 hebras	0.002881	0.058151	2.994939
5 hebras	0.007439	0.057611	5.237664
6 hebras	0.001221	0.037838	7.004997
7 hebras	0.006652	0.037233	6.991640
8 hebras	0.002981	0.055457	3.012973



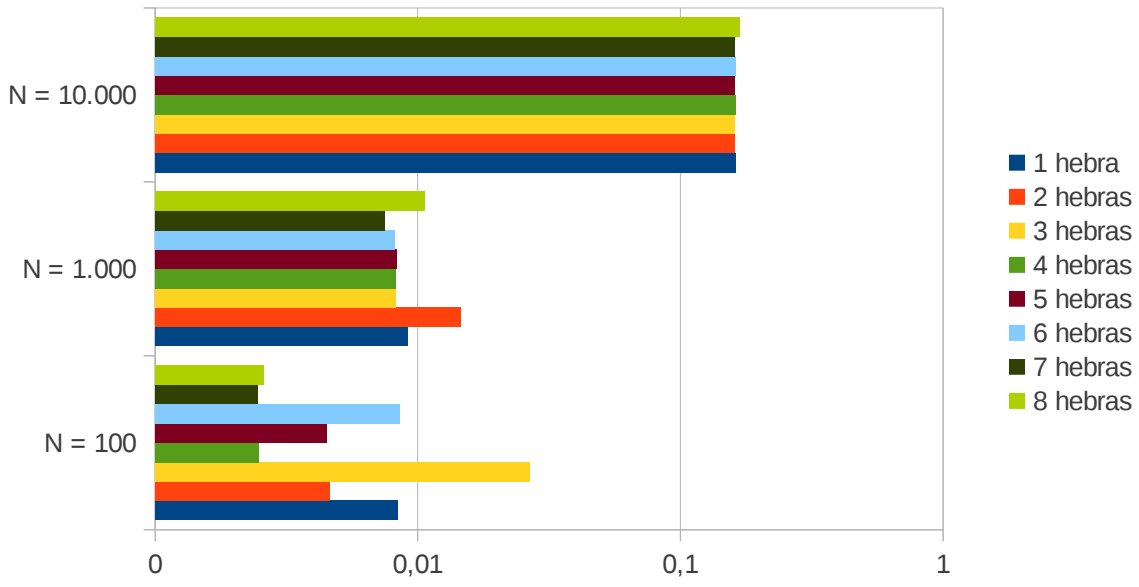
PC-local : pmv-OpenMP-reduction *(paralelización por columnas)*

	N = 100	N = 1.000	N = 10.000
1 hebra	0.000090	0.011491	9.783295
2 hebras	0.000157	0.008950	5.915479
3 hebras	0.002620	0.034220	6.353265
4 hebras	0.003448	0.025919	7.502541



actgrid : pmv-OpenMP-reduction *(paralelización por columnas)*

	N = 100	N = 1.000	N = 10.000
1 hebra	0.008368	0.009135	0.162274
2 hebras	0.004619	0.014569	0.161468
3 hebras	0.026759	0.008263	0.160477
4 hebras	0.002482	0.008244	0.162925
5 hebras	0.004489	0.008327	0.160496
6 hebras	0.008498	0.008169	0.161752
7 hebras	0.002466	0.007501	0.160932
8 hebras	0.002593	0.010659	0.168138



COMENTARIOS SOBRE LOS RESULTADOS:

Tras realizar las gráficas, veamos la mejora máxima de un sistema cuando solo una parte de éste es mejorado.

Podemos ver con claridad que, para tamaños cada vez más grandes, el tiempo de ejecución va creciendo (la ganancia de velocidad es cada vez menor).

Además, cuanto mayor sea el número de hebras que empleemos para ejecutar los programas, la ganancia de velocidad, normalmente, aumenta, es decir, se mejora el rendimiento de un sistema debido a la alteración de uno de sus componentes (aumento del número de hebras), que estará limitada por la fracción de tiempo que utiliza dicho componente.

- Tanto en la paralelización por filas como en la paralelización por columnas (con y sin `reduction`), al ejecutarlo en `PC-local`, nos damos cuenta de que para un tamaño grande (N=10.000) el tiempo de ejecución sale mucho mayor que haciéndolo para tamaños más pequeños (N=100, N=1000).

- Al ejecutar los códigos de paralelización por filas y paralelización por columnas (con `reduction`) en `atcgrid`, podemos ver cómo para tamaños grandes, el tiempo de ejecución también aumenta pero no tanto como en `PC-local`.

No obstante, este tiempo sí aumenta de forma considerable cuando es ejecutado el programa de paralelización por columnas (SIN `reduction`).

Así, podríamos concluir que la paralelización por columnas (CON `reduction`) es la que tarda un tiempo de ejecución menor, y por tanto, la ganancia de velocidad es más notable con respecto a los otros dos códigos.