

Arquitectura de Computadores (AC)

Cuaderno de prácticas.

Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Juan Carlos Ruiz García

Grupo de prácticas: C2

Fecha de entrega: 16/04/2017

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: Lo que ocurre es que cuando incluimos la directiva `default(none)`, el programador (en este caso nosotros) debe especificar el ámbito de todas las variables que intervienen en la región paralela, por lo que el compilador nos pide que definamos el ámbito de la variable `n`.

CÓDIGO FUENTE: `shared-clauseModificado.c`

```
#include <stdio.h>

#ifdef _OPENMP
#include <omp.h>
#endif

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

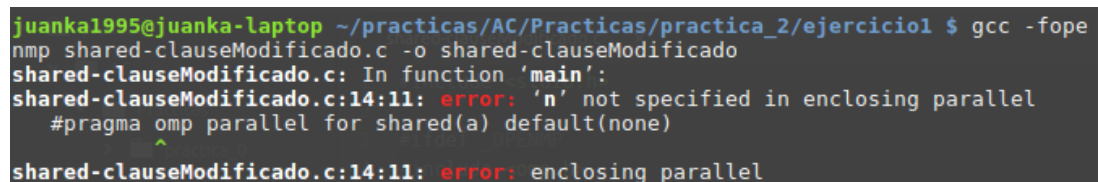
    for (i=0; i<n; i++)
        a[i] = i+1;

    #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:



```
juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio1 $ gcc -fopenmp shared-clauseModificado.c -o shared-clauseModificado
shared-clauseModificado.c: In function 'main':
shared-clauseModificado.c:14:11: error: 'n' not specified in enclosing parallel
    #pragma omp parallel for shared(a) default(none)
    ^
shared-clauseModificado.c:14:11: 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 el valor de *suma* fuera de la región paralela, al entrar en la esta cada hebra tendrá una copia privada sin inicializar de dicha variable, por lo que esa inicialización es inútil para la región paralela y al no inicializar la variable dentro, el valor de *suma* es indeterminado al comenzar la ejecución en paralelo.

Por otra parte si inicializamos la variable dentro de la región paralela, cada copia de cada hebra tendrá un valor concreto al iniciar la ejecución paralela, por lo que el valor de *suma* será correcto.

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 const *argv[]) {
    int i, n = 7;
    int a[n], suma;

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

    suma = 50;

    #pragma omp parallel private(suma)
    {
        // suma = 50;

        printf("Valor inicial de suma: %d\n", suma);

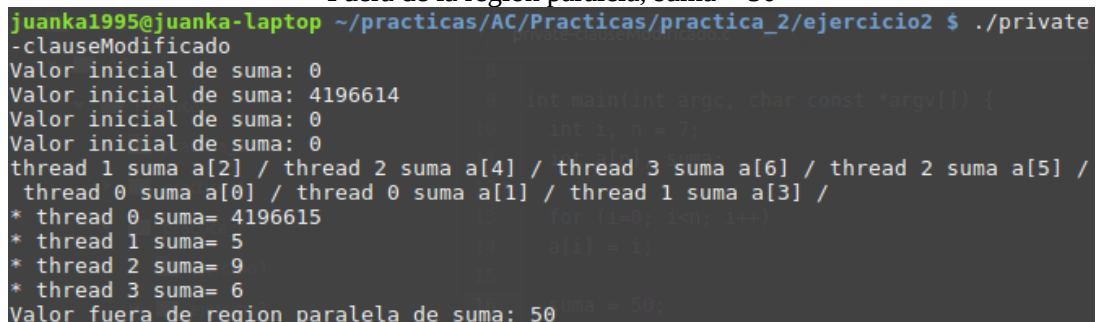
        #pragma omp barrier

        #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("\nValor fuera de region paralela de suma: %d\n", suma);
}
```

CAPTURAS DE PANTALLA:

Fuera de la región paralela, suma = 50



```
juanka1995@juanka-laptop ~/practicass/AC/Practicass/practica_2/ejercicio2 $ ./private-clauseModificado
Valor inicial de suma: 0
Valor inicial de suma: 4196614
Valor inicial de suma: 0
Valor inicial de suma: 0
thread 1 suma a[2] / thread 2 suma a[4] / thread 3 suma a[6] / thread 2 suma a[5] /
thread 0 suma a[0] / thread 0 suma a[1] / thread 1 suma a[3] /
* thread 0 suma= 4196615
* thread 1 suma= 5
* thread 2 suma= 9
* thread 3 suma= 6
Valor fuera de region paralela de suma: 50
```

Dentro de la región paralela, suma = 50

```
juanka1995@juanka-laptop ~/practicass/AC/Practicass/practica_2/ejercicio2 $ ./private
-privateModificado
Valor inicial de suma: 50
Valor inicial de suma: 50
Valor inicial de suma: 50
Valor inicial de suma: 50
thread 2 suma a[4] / thread 1 suma a[2] / thread 1 suma a[3] / thread 2 suma a[5] /
thread 3 suma a[6] / thread 0 suma a[0] / thread 0 suma a[1] /
* thread 0 suma= 51
* thread 3 suma= 56
* thread 2 suma= 59
* thread 1 suma= 55
Valor fuera de region paralela de suma: -16777216
```

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

RESPUESTA: Lo que ocurre es que la variable *suma* pasa a ser *compartida* y por lo tanto todas las hebras modifican la misma variable, es decir, no tienen su propia copia privada. Esto provoca que el resultado sea incierto.

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 const *argv[]) {
    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:

```

juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio3 $ ./private
-closureModificado3
thread 0 suma a[0] / thread 0 suma a[1] / thread 3 suma a[6] / thread 1 suma a[2] /
thread 1 suma a[3] / thread 2 suma a[4] / thread 2 suma a[5] /
* thread 1 suma= 13
* thread 3 suma= 13
* thread 0 suma= 13
* thread 2 suma= 13
juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio3 $ ./private
-closureModificado3
thread 3 suma a[6] / thread 1 suma a[2] / thread 1 suma a[3] / thread 2 suma a[4] /
thread 2 suma a[5] / thread 0 suma a[0] / thread 0 suma a[1] /
* thread 3 suma= 9
* thread 2 suma= 9
* thread 1 suma= 9
* thread 0 suma= 9

```

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: Se debe a que se combina el uso de *firstprivate* y *lastprivate* sobre la variable *suma*. Esto produce que cada hebra tenga una copia privada inicializada a 0 durante su ejecución en paralelo con el resto de hebras (parte del *firstprivate*) y produce además que el valor de la variable *suma* fuera de la región paralela sea el *último* que se guardaría en una ejecución secuencial (parte del *lastprivate*) en vez del *primero* (si solo usáramos *firstprivate*). En este caso, el valor sería *a[6]*.

CAPTURAS DE PANTALLA:

Código con *firstprivate* + *lastprivate*

```

#pragma omp parallel for firstprivate(suma) lastprivate(suma)
for (i=0; i<n; i++)
{
    suma = suma + a[i];
    printf(" thread %d suma a[%d] suma=%d \n",
        omp_get_thread_num(), i, suma);
}
printf("\nFuera de la construcción parallel suma=%d\n", suma);

```

Resultado de varias ejecuciones

```

juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio4 $ ./firstla
stprivate
thread 3 suma a[6] suma=6
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 1 suma a[2] suma=2
thread 1 suma a[3] suma=5
Fuera de la construcción parallel suma=6
juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio4 $ ./firstla
stprivate
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 ocurre si en copyprivate-clause.c se elimina la cláusula copyprivate(a) en la directiva single? ¿A qué cree que es debido?

RESPUESTA: Lo que ocurre es que cada hebra tendrá su propia variable privada *a* y solo modificará su valor la primera hebra que entre a la directiva *single*, siendo el valor de todas las variables *a* indeterminado (ya que ninguna se inicializa), excepto el de la hebra que entra en *single*. Al tener *copyprivate(a)* se realiza una *difusión* del valor de *a* que recoge la primera hebra que llegue a la directiva *single* y se copia al resto de variables *a* privadas de las otras hebras.

CÓDIGO FUENTE: copyprivate-clauseModificado.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 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");

    return 0;
}

```

CAPTURAS DE PANTALLA:

Ejecución sin copyprivate(a)

```

juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio5 $ ./copyprivate-clause
Introduce valor de inicialización a: 10
Single ejecutada por el thread 2
Depués de la región paralela:
b[0] = 4196990 b[1] = 4196990 b[2] = 4196990 b[3] = 0 b[4] = 0 b[5]
] = 10 b[6] = 10 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 resultado con 10 iteraciones sería de 55. Esto se debe a que cada hebra tendrá una copia privada de la variable `suma` inicializada a 0 (según la diapositiva número 26 del Seminario 2) y realizará una suma local de su trozo de región paralela. Una vez terminado lo anterior, cada suma local de cada hebra se sumará con el resto y con la variable `suma` compartida inicialmente con valor 10.

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:

```

juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio6 $ ./reduction-clauseModificado 10
Tras 'parallel' suma=55
juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio6 $ ./reduction-clauseModificado 20
Tras 'parallel' suma=200
juanka1995@juanka-laptop ~/practicac/AC/Practicac/practica_2/ejercicio6 $ ./reduction-clauseModificado 6
Tras 'parallel' suma=25

```

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:**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, suma_aux;

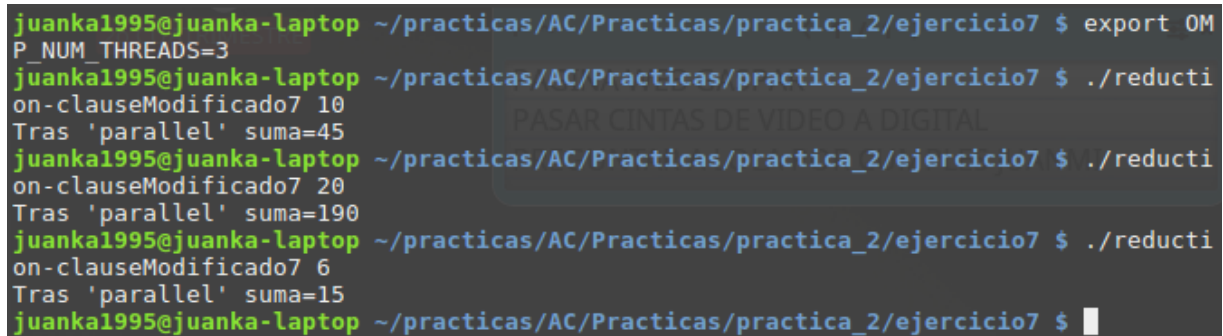
    if(argc < 2) {
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }
    n = atoi(argv[1]); if (n>20) {n=20; printf("n=%d", n);}

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

    #pragma omp parallel private(suma_aux)
    {
        suma_aux = 0;
        #pragma omp for
        for (int i = 0; i < n; i++)
            suma_aux += a[i];
        #pragma omp atomic
        suma += suma_aux;
    }

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

```

CAPTURAS DE PANTALLA:


```

juanka1995@juanka-laptop ~/practicas/AC/Practicas/practica_2/ejercicio7 $ export OMP_NUM_THREADS=3
juanka1995@juanka-laptop ~/practicas/AC/Practicas/practica_2/ejercicio7 $ ./reducti
on-clauseModificado7 10
Tras 'parallel' suma=45
juanka1995@juanka-laptop ~/practicas/AC/Practicas/practica_2/ejercicio7 $ ./reducti
on-clauseModificado7 20
Tras 'parallel' suma=190
juanka1995@juanka-laptop ~/practicas/AC/Practicas/practica_2/ejercicio7 $ ./reducti
on-clauseModificado7 6
Tras 'parallel' suma=15
juanka1995@juanka-laptop ~/practicas/AC/Practicas/practica_2/ejercicio7 $

```

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):

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

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

```
/* pmv-secuencial.c
Producto de una matriz cuadrada por un vector
Para compilar usar (-lrt: real time library):
gcc -O2 pmv-secuencial.c -o pmv-secuencial -lrt
gcc -O2 -S pmv-secuencial.c -lrt //para generar el código ensamblador
Para ejecutar use: pmv-secuencial.c n_filas n_columnas
*/
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
#define PRINTF_ALL // comentar para quitar el printf ...

// que imprime todos los
componentes
//Sólo puede estar definida una de las tres constantes VARIABLES_ (sólo uno de
los ...
//tres defines siguientes puede estar descomentado):
// #define VARIABLES_LOCALES // descomentar para que
las variables sean locales ...

// locales (si se supera el tamaño de la pila
se ...

// generará el error "Violación de Segmento")
// #define VARIABLES_GLOBALES // descomentar para que las variables sean
globales ...

// globales (su longitud no estará limitada por el ...

// tamaño de la pila del programa)
#define VARIABLES_DINAMICAS // descomentar para que las variables sean
dinamicas ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifdef VARIABLES_GLOBALES
#define MAX 8192 //2^13
double M[MAX][MAX], v1[MAX], v2[MAX];
#endif

int main(int argc, char** argv)
{
    int i;
    struct timespec cgt1,cgt2;
    double ncgt; //para tiempo de ejecución
    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2)
    {
        printf("Faltan filas y columnas\n");
        exit(-1);
    }
}
```



```

        unsigned int filas_columnas = atoi(argv[1]);
#ifdef VARIABLES_LOCALES
        double M[filas_columnas][filas_columnas], v1[filas_columnas],
v2[filas_columnas]; // Tamaño variable local en tiempo de ejecución ...
        // disponible en C a partir de actualización C99
#endif
#ifdef VARIABLES_GLOBALES
        if (filas_columnas>MAX) filas_columnas=MAX;
#endif
#ifdef VARIABLES_DINAMICAS
        double **M, *v1, *v2;
        M = (double **) malloc(filas_columnas*sizeof(double*));
        v1 = (double*) malloc(filas_columnas*sizeof(double)); // malloc
necesita el tamaño en bytes
        v2 = (double*) malloc(filas_columnas*sizeof(double)); //si no
hay espacio suficiente malloc devuelve NULL
        if ( (v1==NULL) || (v2==NULL) || (M==NULL)){
            printf("Error en la reserva de espacio para los vectores\n");
            exit(-2);
        }
#endif

        //Inicialiar v1 y M
        for(i=0; i<filas_columnas; i++){
            v1[i] = filas_columnas*0.1-i*0.1;//los valores
dependen de filas_columnas
            v2[i] = 0.0;
            M[i] = (double*)
malloc(filas_columnas*sizeof(double));
            for (int j = 0; j < filas_columnas; j++) {
                M[i][j] = filas_columnas*0.1+i*0.1;
            }
        }

        clock_gettime(CLOCK_REALTIME,&cgt1);
        //Calcular producto de v1 por M, en v2
        for (int i = 0; i < filas_columnas; i++) {
            for (int j = 0; j < filas_columnas; j++) {
                v2[i] += M[i][j] * v1[j];
            }
        }
        clock_gettime(CLOCK_REALTIME,&cgt2);
        ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+ (double)
((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
        //Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
        printf("Tiempo(seg.):%11.9f\t Tamaño vectores y matriz:
%u\n",ncgt,filas_columnas);
        for(i=0; i<filas_columnas; i++)
            printf("V2[%d]=(%8.6f) \n", i,v2[i]);
#else
        printf("Tiempo(seg.):%11.9f\t Tamaño vectores y matriz:%u\t
V2[0]=(%8.6f) V2[%d]=(%8.6f) \n", ncgt,filas_columnas,v2[0],filas_columnas-
1,v2[filas_columnas-1]);
#endif
#ifdef VECTOR_DYNAMIC
        free(v1); // libera el espacio reservado para v1
        free(v2); // libera el espacio reservado para v2
        free(*M); // libera el espacio reservado para v3
#endif

        return 0;
}

```

CAPTURAS DE PANTALLA:

```

juanka1995@juanka-laptop ~/practicass/AC/Practicass/practica_2/ejercicio8 $ gcc -O2 pmv-secuencial.c -o pmv-secuencial -lrt
juanka1995@juanka-laptop ~/practicass/AC/Practicass/practica_2/ejercicio8 $ ./pmv-secuencial 4
llego aqui
Tiempo(seg.):0.000000318      Tamaño vectores y matriz:4
V2[0]=(0.400000)
V2[1]=(0.500000)
V2[2]=(0.600000)
V2[3]=(0.700000)

```

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

```

/* pmv-secuencial.c
Producto de una matriz cuadrada por un vector
Para compilar usar (-lrt: real time library):
gcc -O2 pmv-secuencial-a.c -o pmv-secuencial-a -lrt -fopenmp
gcc -O2 -S pmv-secuencial-a.c -lrt -fopenmp //para generar el código
ensamblador
Para ejecutar use: pmv-secuencial-a.c n_filas n_columnas
*/
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif
// #define PRINTF_ALL// comentar para quitar el printf ...

// que imprime todos los
componentes
//Sólo puede estar definida una de las tres constantes VARIABLES_ (sólo uno de
los ...
//tres defines siguientes puede estar descomentado):
// #define VARIABLES_LOCALES // descomentar para que

```

```

las variables sean locales ...

// locales (si se supera el tamaño de la pila
se ...

// generará el error "Violación de Segmento")
// #define VARIABLES_GLOBALES // descomentar para que las variables sean
globales ...

// globales (su longitud no estará limitada por el ...

// tamaño de la pila del programa)
#define VARIABLES_DINAMICAS // descomentar para que las variables sean
dinamicas ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifdef VARIABLES_GLOBALES
#define MAX 8192 //2^13
double M[MAX][MAX], v1[MAX], v2[MAX];
#endif

int main(int argc, char** argv)
{
    int i;
    struct timespec cgt1,cgt2;
    double ncgt; //para tiempo de ejecución
    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2)
    {
        printf("Faltan filas y columnas\n");
        exit(-1);
    }

    unsigned int filas_columnas = atoi(argv[1]);
#ifdef VARIABLES_LOCALES
    double M[filas_columnas][filas_columnas], v1[filas_columnas],
v2[filas_columnas]; // Tamaño variable local en tiempo de ejecución ...
// disponible en C a partir de actualización C99
#endif
#ifdef VARIABLES_GLOBALES
    if (filas_columnas>MAX) filas_columnas=MAX;
#endif
#ifdef VARIABLES_DINAMICAS
    double **M, *v1, *v2;
    M = (double **) malloc(filas_columnas*sizeof(double*));
    v1 = (double*) malloc(filas_columnas*sizeof(double)); // malloc
necesita el tamaño en bytes
    v2 = (double*) malloc(filas_columnas*sizeof(double)); //si no
hay espacio suficiente malloc devuelve NULL
    if ( (v1==NULL) || (v2==NULL) || (M==NULL)){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }
#endif
#ifdef _OPENMP
    //Inicialiar v1 y M
    for(i=0; i<filas_columnas; i++){
        v1[i] = filas_columnas*0.1-i*0.1;//los valores
dependen de filas_columnas

```

```

        v2[i] = 0.0;
        M[i] = (double*)
malloc(filas_columnas*sizeof(double));
        for (int j = 0; j < filas_columnas; j++) {
            M[i][j] = filas_columnas*0.1+i*0.1;
        }
    }
    clock_gettime(CLOCK_REALTIME,&cgt1);
    //Calcular producto de v1 por M, en v2
    for (int i = 0; i < filas_columnas; i++) {
        for (int j = 0; j < filas_columnas; j++) {
            v2[i] += M[i][j] * v1[j];
        }
    }
    clock_gettime(CLOCK_REALTIME,&cgt2);
#else
    #pragma omp parallel
    {
        // Inicializar v1 y M
        #pragma omp for
        for(i=0; i<filas_columnas; i++){
            v1[i] = filas_columnas*0.1-i*0.1;//los
valores dependen de filas_columnas
            v2[i] = 0.0;
            M[i] = (double*)
malloc(filas_columnas*sizeof(double));
            for (int j = 0; j < filas_columnas; j+
+) {
                M[i][j] =
filas_columnas*0.1+i*0.1;
            }
            clock_gettime(CLOCK_REALTIME,&cgt1);
            #pragma omp for
            for (int i = 0; i < filas_columnas; i++) {
                for (int j = 0; j < filas_columnas; j+
+) {
                    v2[i] += M[i][j] *
v1[j];
                }
            }
            clock_gettime(CLOCK_REALTIME,&cgt2);
        }
    }
#endif

    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+ (double)
((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
    //Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
    printf("Tiempo(seg.):%11.9f\t Tamaño vectores y matriz:
%u\n",ncgt,filas_columnas);
    for(i=0; i<filas_columnas; i++)
        printf("V2[%d]=(%8.6f) \n", i,v2[i]);
#else
    printf("Tiempo(seg.):%11.9f\t Tamaño vectores y matriz:%u\t
V2[0]=(%8.6f) V2[%d]=(%8.6f) \n", ncgt,filas_columnas,v2[0],filas_columnas-
1,v2[filas_columnas-1]);
#endif
#ifdef VECTOR_DYNAMIC
    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2

```

```

        free(*M); // libera el espacio reservado para v3
    #endif
        return 0;
}

```

CÓDIGO FUENTE: pmv-OpenMP-b.c

```

/* pmv-OpenMP-b.c
Producto de una matriz cuadrada por un vector
Para compilar usar (-lrt: real time library):
gcc -O2 pmv-OpenMP-b.c -o pmv-OpenMP-b -lrt -fopenmp
gcc -O2 -S pmv-OpenMP-b.c -lrt -fopenmp //para generar el código ensamblador
Para ejecutar use: pmv-OpenMP-b.c n_filas n_columnas
*/
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif
// #define PRINTF_ALL// comentar para quitar el printf ...

// que imprime todos los
componentes
//Sólo puede estar definida una de las tres constantes VARIABLES_ (sólo uno de
los ...
//tres defines siguientes puede estar descomentado):
// #define VARIABLES_LOCALES // descomentar para que
las variables sean locales ...

// locales (si se supera el tamaño de la pila
se ...

// generará el error "Violación de Segmento")
// #define VARIABLES_GLOBALES // descomentar para que las variables sean
globales ...

// globales (su longitud no estará limitada por el ...

// tamaño de la pila del programa)
#define VARIABLES_DINAMICAS // descomentar para que las variables sean
dinamicas ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifdef VARIABLES_GLOBALES
#define MAX 8192 //2^13
double M[MAX][MAX], v1[MAX], v2[MAX];
#endif

int main(int argc, char** argv)
{
    int i;
    struct timespec cgt1,cgt2;
    double ncgt; //para tiempo de ejecución
    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2)
    {
        printf("Faltan filas y columnas\n");
    }
}

```

```

        exit(-1);
    }

    unsigned int filas_columnas = atoi(argv[1]);
#ifdef VARIABLES_LOCALES
    double M[filas_columnas][filas_columnas], v1[filas_columnas],
v2[filas_columnas]; // Tamaño variable local en tiempo de ejecución ...
    // disponible en C a partir de actualización C99
#endif
#ifdef VARIABLES_GLOBALES
    if (filas_columnas>MAX) filas_columnas=MAX;
#endif
#ifdef VARIABLES_DINAMICAS
    double **M, *v1, *v2;
    M = (double **) malloc(filas_columnas*sizeof(double*));
    v1 = (double*) malloc(filas_columnas*sizeof(double)); // malloc
necesita el tamaño en bytes
    v2 = (double*) malloc(filas_columnas*sizeof(double)); //si no
hay espacio suficiente malloc devuelve NULL
    if ( (v1==NULL) || (v2==NULL) || (M==NULL)){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }
#endif
#ifdef _OPENMP
    //Inicialiar v1 y M
    for(i=0; i<filas_columnas; i++){
        v1[i] = filas_columnas*0.1-i*0.1;//los valores
dependen de filas_columnas
        v2[i] = 0.0;
        M[i] = (double*)
malloc(filas_columnas*sizeof(double));
        for (int j = 0; j < filas_columnas; j++) {
            M[i][j] = filas_columnas*0.1+i*0.1;
        }
    }
    clock_gettime(CLOCK_REALTIME,&cgt1);
    //Calcular producto de v1 por M, en v2
    for (int i = 0; i < filas_columnas; i++) {
        for (int j = 0; j < filas_columnas; j++) {
            v2[i] += M[i][j] * v1[j];
        }
    }
    clock_gettime(CLOCK_REALTIME,&cgt2);
#else
    #pragma omp parallel
    {
        // Inicializar v1 y M
        #pragma omp for
        for(i=0; i<filas_columnas; i++){
            v1[i] = filas_columnas*0.1-i*0.1;//los
valores dependen de filas_columnas
            v2[i] = 0.0;
            M[i] = (double*)
malloc(filas_columnas*sizeof(double));
            for (int j = 0; j < filas_columnas; j+
+ ) {
                M[i][j] =
filas_columnas*0.1+i*0.1;
            }
        }
    }
#endif
}

```

```

        clock_gettime(CLOCK_REALTIME,&cgt1);
        for (int i = 0; i < filas_columnas; i++) {
            #pragma omp for
                for (int j = 0; j < filas_columnas; j+
+) {
                                v2[i] += M[i][j] *
v1[j];
                                }
        }
        clock_gettime(CLOCK_REALTIME,&cgt2);
    }
#endif

    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+ (double)
    ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
    //Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
    printf("Tiempo(seg.):%11.9f\t Tamaño vectores y matriz:
%u\n",ncgt,filas_columnas);
    for(i=0; i<filas_columnas; i++)
        printf("V2[%d]=(%8.6f) \n", i,v2[i]);
#else
    printf("Tiempo(seg.):%11.9f\t Tamaño vectores y matriz:%u\t
V2[0]=(%8.6f) V2[%d]=(%8.6f) \n", ncgt,filas_columnas,v2[0],filas_columnas-
1,v2[filas_columnas-1]);
#endif
#ifdef VECTOR_DYNAMIC
    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
    free(*M); // libera el espacio reservado para v3
#endif

    return 0;
}

```

RESPUESTA: Por suerte, no he sufrido ningún error a la hora de la compilación ni de la ejecución de mi programa A, sin embargo en el B no sufrí problemas de complicación pero sí de ejecución ya que me daba resultados distintos a los secuenciales. Lo solucioné haciendo atómicas las escrituras en V2 mediante la clausula *critical*.

CAPTURAS DE PANTALLA:

```

juanka1995@juanka-laptop ~/practicass/AC/Practicass/practica_2/ejercicio9 $ gcc -O2 pmv-OpenMP-b.c -o pmv-OpenMP-b -lrt -fopenmp
pmv-OpenMP-b.c: In function 'main':
pmv-OpenMP-b.c:98:13: error: work-sharing region may not be closely nested inside of work-sharing, critical, ordered, master or
explicit task region
    #pragma omp for
    ^

```

```

juanka1995@juanka-laptop ~/practicass/AC/Practicass/practica_2/ejercicio9 $ ./pmv-OpenMP-b 4
Tiempo(seg.):0.000005345 Tamaño vectores y matriz:4
V2[0]=(0.400000)
V2[1]=(0.500000)
V2[2]=(0.600000)
V2[3]=(0.700000)
juanka1995@juanka-laptop ~/practicass/AC/Practicass/practica_2/ejercicio9 $ gcc -O2 pmv-OpenMP-a.c -o pmv-OpenMP-a
juanka1995@juanka-laptop ~/practicass/AC/Practicass/practica_2/ejercicio9 $ ./pmv-OpenMP-a 4
Tiempo(seg.):0.002301309 Tamaño vectores y matriz:4
V2[0]=(0.400000)
V2[1]=(0.500000)
V2[2]=(0.600000)
V2[3]=(0.700000)

```

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

```
/* pmv-OpenmMP-reduction.c
Producto de una matriz cuadrada por un vector
Para compilar usar (-lrt: real time library):
gcc -O2 pmv-OpenmMP-reduction.c -o pmv-OpenmMP-reduction -lrt -fopenmp
gcc -O2 -S pmv-OpenmMP-reduction.c -lrt -fopenmp //para generar el código
ensamblador
Para ejecutar use: pmv-OpenmMP-reduction.c n_filas n_columnas
*/
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif
#define PRINTF_ALL // comentar para quitar el printf ...

// que imprime todos los
componentes
//Sólo puede estar definida una de las tres constantes VARIABLES_ (sólo uno de
los ...
//tres defines siguientes puede estar descomentado):
// #define VARIABLES_LOCALES // descomentar para que
las variables sean locales ...

// locales (si se supera el tamaño de la pila
se ...

// generará el error "Violación de Segmento")
// #define VARIABLES_GLOBALES // descomentar para que las variables sean
globales ...

// globales (su longitud no estará limitada por el ...

// tamaño de la pila del programa)
#define VARIABLES_DINAMICAS // descomentar para que las variables sean
dinamicas ...
// dinámicas (memoria reutilizable durante la ejecución)
#ifdef VARIABLES_GLOBALES
#define MAX 8192 //2^13
double M[MAX][MAX], v1[MAX], v2[MAX];
#endif

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



```

int i;
struct timespec cgt1,cgt2;
double ncgt; //para tiempo de ejecución
//Leer argumento de entrada (nº de componentes del vector)
if (argc<2)
{
    printf("Faltan filas y columnas\n");
    exit(-1);
}

unsigned int filas_columnas = atoi(argv[1]);
#ifdef VARIABLES_LOCALES
double M[filas_columnas][filas_columnas], v1[filas_columnas],
v2[filas_columnas]; // Tamaño variable local en tiempo de ejecución ...
// disponible en C a partir de actualización C99
#endif
#ifdef VARIABLES_GLOBALES
if (filas_columnas>MAX) filas_columnas=MAX;
#endif
#ifdef VARIABLES_DINAMICAS
double **M, *v1, *v2;
M = (double **) malloc(filas_columnas*sizeof(double*));
v1 = (double*) malloc(filas_columnas*sizeof(double)); // malloc
necesita el tamaño en bytes
v2 = (double*) malloc(filas_columnas*sizeof(double)); //si no
hay espacio suficiente malloc devuelve NULL
if ( (v1==NULL) || (v2==NULL) || (M==NULL)){
    printf("Error en la reserva de espacio para los vectores\n");
    exit(-2);
}
#endif
#ifdef _OPENMP
//Inicialiar v1 y M
for(i=0; i<filas_columnas; i++){
    v1[i] = filas_columnas*0.1-i*0.1;//los valores
dependen de filas_columnas
    v2[i] = 0.0;
    M[i] = (double*)
malloc(filas_columnas*sizeof(double));
    for (int j = 0; j < filas_columnas; j++) {
        M[i][j] = filas_columnas*0.1+i*0.1;
    }
}
clock_gettime(CLOCK_REALTIME,&cgt1);
//Calcular producto de v1 por M, en v2
for (int i = 0; i < filas_columnas; i++) {
    for (int j = 0; j < filas_columnas; j++) {
        v2[i] += M[i][j] * v1[j];
    }
}
clock_gettime(CLOCK_REALTIME,&cgt2);
#else
// Inicializar v1 y M
#pragma omp parallel for
for(i=0; i<filas_columnas; i++){
    v1[i] = filas_columnas*0.1-i*0.1;//los
valores dependen de filas_columnas
    v2[i] = 0.0;
    M[i] = (double*)
malloc(filas_columnas*sizeof(double));
    for (int j = 0; j < filas_columnas; j+
+) {

```

```

M[i][j] =
filas_columnas*0.1+i*0.1;
    }
    }
    clock_gettime(CLOCK_REALTIME,&cgt1);

    for (int i = 0; i < filas_columnas; i++) {
        double suma = 0.0;
        #pragma omp parallel for reduction(+:suma)
        for (int j = 0; j < filas_columnas; j+
+) {
            suma += M[i][j] * v1[j];
        }
        v2[i] = suma;
    }
    clock_gettime(CLOCK_REALTIME,&cgt2);
#endif

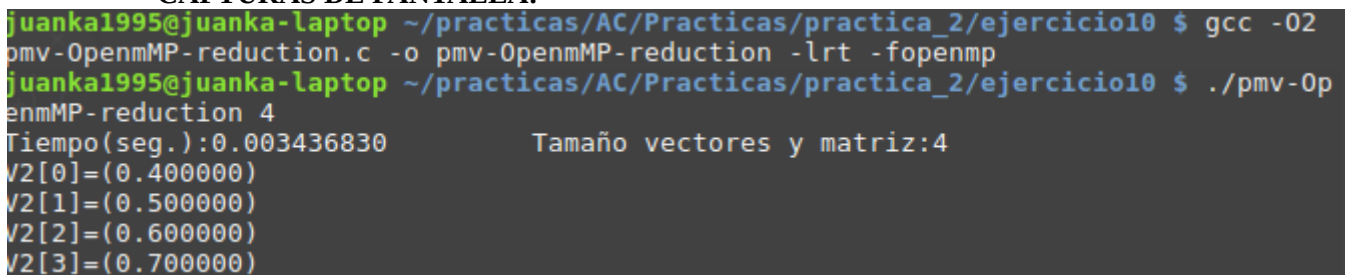
    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+ (double)
((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
    //Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
    printf("Tiempo(seg.):%11.9f\t Tamaño vectores y matriz:
%u\n",ncgt,filas_columnas);
    for(i=0; i<filas_columnas; i++)
        printf("V2[%d]=(%8.6f) \n", i,v2[i]);
#else
    printf("Tiempo(seg.):%11.9f\t Tamaño vectores y matriz:%u\t
V2[0]=(%8.6f) V2[%d]=(%8.6f) \n", ncgt,filas_columnas,v2[0],filas_columnas-
1,v2[filas_columnas-1]);
#endif
#ifdef VECTOR_DYNAMIC
    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
    free(*M); // libera el espacio reservado para v3
#endif

    return 0;
}

```

RESPUESTA: No he tenido ningún problema para la compilación ni para la ejecución. Solo ha sido necesario usar una variable auxiliar llamada *suma* para utilizarla con la cláusula *reduction* y posteriormente igualar $v[i]=suma$.

CAPTURAS DE PANTALLA:



```

juanka1995@juanka-laptop ~/practicas/AC/Practicas/practica_2/ejercicio10 $ gcc -O2
pmv-OpenmMP-reduction.c -o pmv-OpenmMP-reduction -lrt -fopenmp
juanka1995@juanka-laptop ~/practicas/AC/Practicas/practica_2/ejercicio10 $ ./pmv-Op
enmMP-reduction 4
Tiempo(seg.):0.003436830          Tamaño vectores y matriz:4
V2[0]=(0.400000)
V2[1]=(0.500000)
V2[2]=(0.600000)
V2[3]=(0.700000)

```

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):

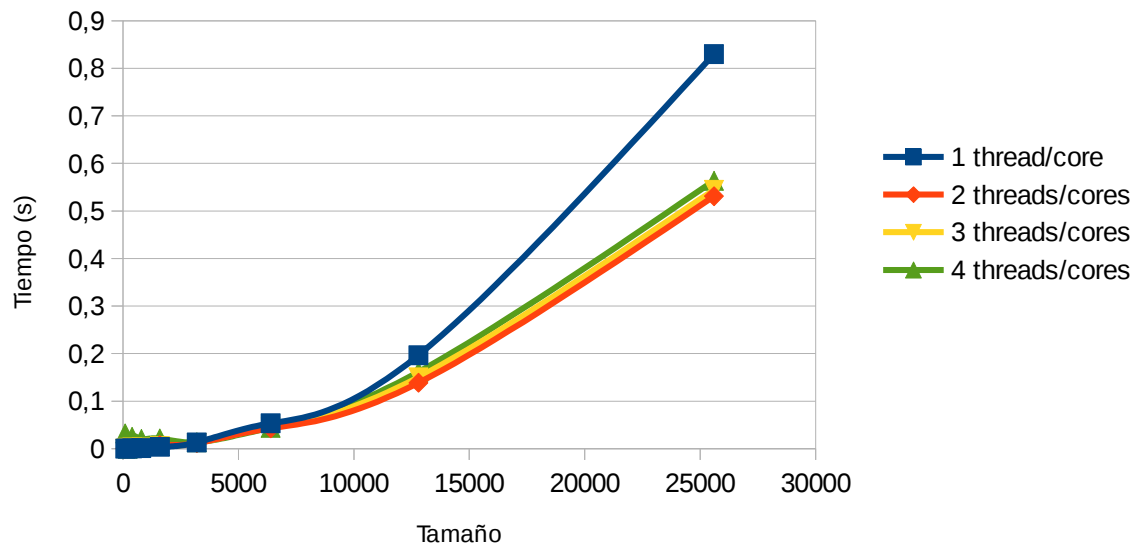
PC LOCAL

Intel(R) Core(TM) i5-4210U CPU @ 1,70GHz

Tamaño	1 thread/core	2 threads/cores	3 threads/cores	4 threads/cores
100	0,000076025	0,000078221	0,000257411	0,032069327
200	0,000147541	0,000296465	0,001877636	0,000259539
400	0,000784843	0,00130877	0,002401736	0,024778593
800	0,001132628	0,001141847	0,002067621	0,020070399
1600	0,003810957	0,007290621	0,005145536	0,021505021
3200	0,013132454	0,011600327	0,011748461	0,013530481
6400	0,053210113	0,0429367	0,045212094	0,043962183
12800	0,196697837	0,138652746	0,150143649	0,161542046
25600	0,830233147	0,530919822	0,544274846	0,563764319

PC Local

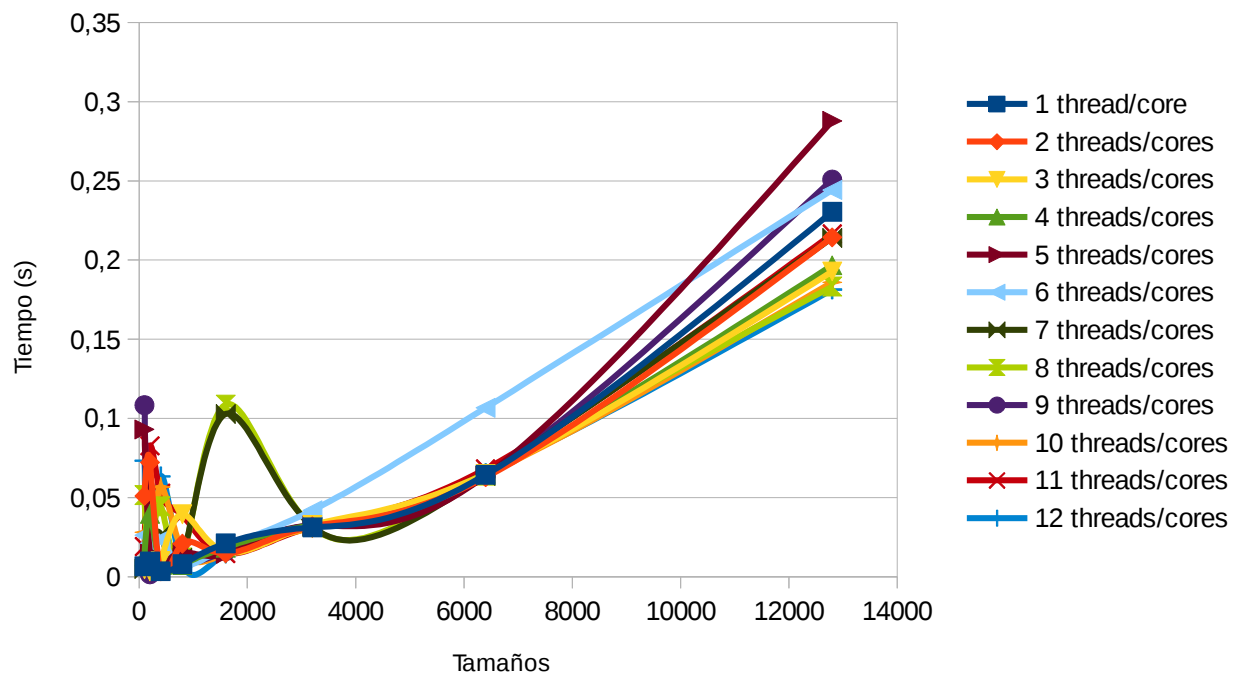
Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz



Atcgrid

Tamaño	1 thread/core	2 threads/cores	3 threads/cores	4 threads/cores	5 threads/cores	6 threads/cores
100	0,006608535	0,051057469	0,004785148	0,012906943	0,093064782	0,026264198
200	0,009675987	0,072307194	0,003435944	0,039718151	0,014591151	0,021188838
400	0,003516009	0,003477759	0,010091431	0,007447015	0,004700741	0,023482892
800	0,007950755	0,020879756	0,039562929	0,007446369	0,015499262	0,007716316
1600	0,021120492	0,014818406	0,015722119	0,018507499	0,014558127	0,018390392
3200	0,031257781	0,031876655	0,032703285	0,032698316	0,032349084	0,041976697
6400	0,064454092	0,062894141	0,064919968	0,06533655	0,063771158	0,106728379
12800	0,230501116	0,214322585	0,192836212	0,19672242	0,287934568	0,244268742

Tamaño	7 threads/cores	8 threads/cores	9 threads/cores	10 threads/cores	11 threads/cores	12 threads/cores
100	0,004950604	0,051657917	0,108384714	0,028119983	0,019137292	0,073346134
200	0,012815629	0,012289057	0,001637188	0,016109146	0,082702963	0,00320634
400	0,02673448	0,044551672	0,003686464	0,056775663	0,052713138	0,063427024
800	0,011928936	0,012128165	0,007734607	0,016805219	0,038782314	0,008119522
1600	0,102834751	0,108807561	0,016224554	0,014756974	0,01468134	0,014785913
3200	0,031764655	0,031502008	0,032463716	0,030695677	0,031660572	0,031691387
6400	0,063807202	0,063926866	0,064028716	0,064471113	0,068106318	0,065512657
12800	0,214059687	0,183314715	0,250826456	0,185994312	0,216459624	0,1814082

Atcgrid**COMENTARIOS SOBRE LOS RESULTADOS:**

Los resultados obtenidos en mi ordenador me resultan correctos y convincentes, sin embargo en atcgrid hay algunos picos de la gráfica que no entiendo y algunos casos en los que con menos threads tarda menos que en otros casos con más threads.