2º curso / 2º cuatr.

Grado Ing. Inform.

Doble Grado Ing. Inform. y Mat.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 1. Programación paralela I: Directivas OpenMP

Estudiante (nombre y apellidos): Marta Gómez Macías
Grupo de prácticas: C1
Fecha de entrega: 01/04/2014
Fecha evaluación en clase:

. Usar la directiva `parallel` combinada con directivas de trabajo compartido en los ejemplos `bucle-for.c` y `sections.c` del seminario. Incorporar el código fuente resultante al cuaderno de prácticas.

RESPUESTA: código fuente `bucle-forModificado.c`

```
#pragma omp parallel for
    for(i = 0; i < n; i++)
        printf("thread %d ejecuta la iteración %d del bucle\n",
          omp_get_thread_num(),i);
```

RESPUESTA: código fuente `sectionsModificado.c`

```
#pragma omp parallel sections
{
    #pragma omp section
        (void)funcA();
    #pragma omp section
        (void)funcB();
}
```

. Imprimir los resultados del programa `single.c` usando una directiva `single` dentro de la construcción `parallel` en lugar de imprimirlos fuera de la región `parallel`. Añadir lo necesario, dentro de la nueva directiva `single` incorporada, para que se imprima el identificador del thread que ejecuta el bloque estructurado de la directiva `single`. Incorpore en su cuaderno de trabajo el código fuente y volcados de pantalla con los resultados de ejecución obtenidos.

RESPUESTA: código fuente `singleModificado.c`

```
#pragma omp parallel
{
    #pragma omp single
    {
        printf("Introduce valor de inicializacion a:\n");
        scanf("%d",&a);
        printf("Single ejecutado por el thread %d\n",
              omp_get_thread_num());
    }

    #pragma omp for
    for (i=0;i<n;i++)
```

```
        b[i]=a;

    #pragma omp single
    {
        printf("Dentro de un single en paralel,ejecutado por la hebra %d:\n",
              omp_get_thread_num());
        for (i=0;i<n;i++) printf("b[%d]=%d\t", i,b[i]);
        printf("\n");
    }
}
```

CAPTURAS DE PANTALLA:

```
[marta@marta-PC Ejercicio2]$ ./singlemodificado
Introduce valor de inicializacion a:
5
Single ejecutado por el thread 0
Dentro de un single en paralel, ejecutado por la hebra 1:
b[0]=5  b[1]=5  b[2]=5  b[3]=5  b[4]=5  b[5]=5  b[6]=5  b[7]=5  b[8]=5
[marta@marta-PC Ejercicio2]$ ./singlemodificado
Introduce valor de inicializacion a:
5157
Single ejecutado por el thread 1
Dentro de un single en paralel, ejecutado por la hebra 3:
b[0]=5157       b[1]=5157       b[2]=5157       b[3]=5157       b[4]=5157       b[5]=5157       b[6]=5157       b[7]=5157       b[8]=5157
[marta@marta-PC Ejercicio2]$
```

. Imprimir los resultados del programa `single.c` usando una directiva `master` dentro de la construcción `parallel` en lugar de imprimirlos fuera de la región `parallel`. Añadir lo necesario, dentro de la nueva directiva `master` incorporada, para que se imprima el identificador del thread que ejecuta el bloque estructurado de la directiva `master`. Incorpore en su cuaderno el código fuente y volcados de pantalla con los resultados de ejecución obtenidos. ¿Qué diferencia observa con respecto a los resultados de ejecución del ejercicio anterior?

RESPUESTA: código fuente `singleModificado2.c`

```
#pragma omp parallel
{
    #pragma omp single
    {
        printf("Introduce valor de inicializacion a:\n");
        scanf("%d",&a);
        printf("Single ejecutado por el thread %d\n",
            omp_get_thread_num());
    }


    #pragma omp for
    for (i=0;i<n;i++)
        b[i]=a;


    #pragma omp master
    {
        printf("Dentro de un master en paralel,ejecutado por la hebra %d:\n",
            omp_get_thread_num());
        for (i=0;i<n;i++) printf("b[%d]=%d\t", i,b[i]);
        printf("\n");
    }
}
```

CAPTURAS DE PANTALLA:

```
[marta@marta-PC Ejercicio3]$ ./singlemodificado2
Introduce valor de inicializacion a:
55
Single ejecutado por el thread 2
Dentro de un master en paralel, ejecutado por la hebra 0:
b[0]=55 b[1]=55 b[2]=55 b[3]=55 b[4]=55 b[5]=55 b[6]=55 b[7]=55 b[8]=55
[marta@marta-PC Ejercicio3]$ ./singlemodificado2
Introduce valor de inicializacion a:
47
Single ejecutado por el thread 0
Dentro de un master en paralel, ejecutado por la hebra 0:
b[0]=47 b[1]=47 b[2]=47 b[3]=47 b[4]=47 b[5]=47 b[6]=47 b[7]=47 b[8]=47
[marta@marta-PC Ejercicio3]$ ./singlemodificado2
Introduce valor de inicializacion a:
88
Single ejecutado por el thread 0
Dentro de un master en paralel, ejecutado por la hebra 0:
b[0]=88 b[1]=88 b[2]=88 b[3]=88 b[4]=88 b[5]=88 b[6]=88 b[7]=88 b[8]=88
[marta@marta-PC Ejercicio3]$
```

RESPUESTA A LA PREGUNTA: que los resultados siempre los imprime la hebra 0, es decir, la hebra master, mientras que en el ejercicio anterior podía imprimir los resultados cualquier hebra (incluida la master).

. ¿Por qué si se elimina directiva `barrier` en el ejemplo `master.c` la suma que se calcula e imprime no siempre es correcta? Responda razonadamente.

RESPUESTA: porque si no se espera a que todas las hebras hayan sumado su resultado a la variable **suma**, puede que la ejecución termine sin haber sumado las sumas locales de todas las hebras y dando, por tanto, un resultado erróneo.

. El programa secuencial C del Listado 1 calcula la suma de dos vectores (v3 = v1 + v2; v3(i) = v1(i) + v2(i), i=0,...N-1). Generar el ejecutable del programa del Listado 1 para **vectores globales**. Usar `time` (Lección 3/ Tema 1) en la línea de comandos para obtener, en el PC local, el tiempo de ejecución (*elapsed time*) y el tiempo de CPU del usuario y del sistema generado. Obtenga los tiempos para vectores con 10000000 componentes. ¿La suma de los tiempos de CPU del usuario y del sistema es mayor o igual que el tiempo real (*elapsed*)? Justifique la respuesta.

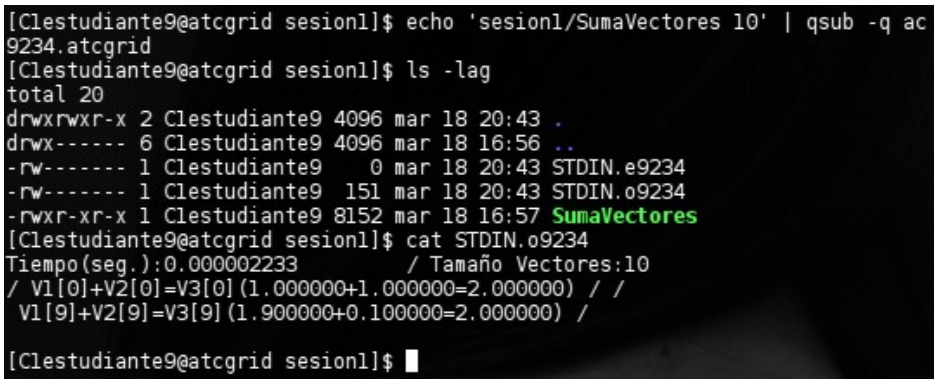
CAPTURAS DE PANTALLA:

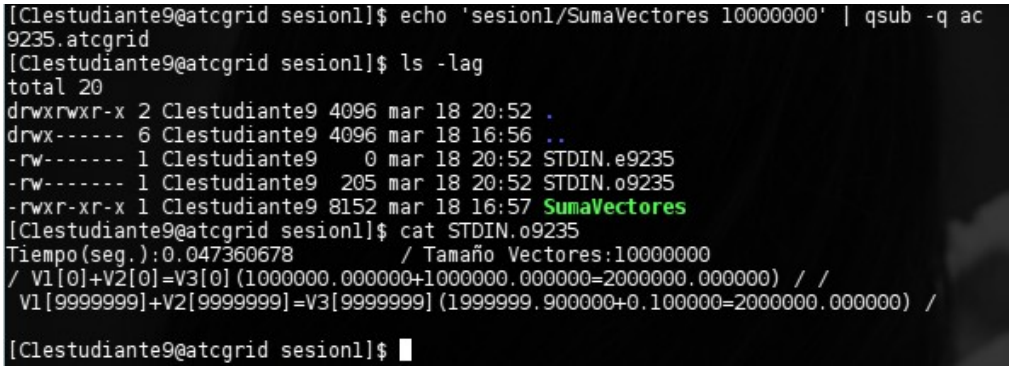
```
[marta@marta-PC Ejercicio4]$ gcc -O2 sumaVectoresC.c -o SumaVectores -lrt
[marta@marta-PC Ejercicio4]$ time ./SumaVectores 10000000
Tiempo(seg.):0.036898548        / Tamaño Vectores:10000000
/ V1[0]+V2[0]=V3[0](1000000.000000+1000000.000000=2000000.000000) / /
 V1[9999999]+V2[9999999]=V3[9999999](1999999.900000+0.100000=2000000.000000) /


real    0m0.115s
user    0m0.090s
sys     0m0.020s
```

El CPUtime = user + sys es el tiempo que tardan en ejecutar el código correspondiente a llamadas al sistema y el código correspondiente al código del programa. El tiempo de ejecución real/elapsed también incluye esperas de E/S, que no forman parte del CPU time. Por eso el tiempo real es mayor o igual al de CPU.

. Generar el código ensamblador a partir del programa secuencial C del Listado 1 para **vectores globales** (para generar el código ensamblador tiene que compilar usando `-S` en lugar de `-o`). Utilice el fichero con el código fuente ensamblador generado y el fichero ejecutable generado en el ejercicio 5 para obtener para atcgrid los MIPS (*Millions of Instructions Per Second*) y los MFLOPS (*Millions of FLOating-point Per Second*) del código que obtiene la suma de vectores (código entre las funciones `clock_gettime()`); el cálculo se debe hacer para 10 y 10000000 componentes en los vectores (consulte la Lección 3/Tema1 AC). Incorpore **el código ensamblador de la parte de la suma de vectores** en el cuaderno.

CAPTURAS DE PANTALLA:





RESPUESTA: Cuando ejecutamos el programa para 10 componentes del vector, obtenemos un tiempo de 0.000002233 segundos. En el código ensamblador tenemos 7 instrucciones fuera del bucle (incluidas las llamadas a clock_gettime) y dentro del bucle tenemos 6 instrucciones, que multiplicadas por el número de iteraciones, 10 en este caso, nos dan un total de 60 instrucciones. Por tanto, en nuestro código tenemos 67 instrucciones que tardan en ejecutarse 0.000002233 segundos, por tanto:

$$MIPS = \frac{67}{0{,}000002233 \cdot 10^6} = 30{,}00448$$

En atcgrid se ejecutan 30 millones de instrucciones por segundo.

De esas 67 instrucciones, 30 son de coma flotante, por tanto:

$$MFLOPS = \frac{30}{0{,}000002233 \cdot 10^6} = 13{,}43484$$

En atcgrid se ejecutan 13 millones de operaciones en coma flotante por segundo.

RESPUESTA: Cuando ejecutamos el programa para 10 millones de componentes del vector, obtenemos un tiempo de 0.047360678 segundos. Por tanto, si aplicamos las mismas fórmulas de antes cambiando el tiempo de ejecución y el número de instrucciones (ya que éste depende del número de componentes del vector) obtenemos que:

$$MIPS = \frac{60.000.007}{0{,}047360678 \cdot 10^6} = 1266{,}873903$$

$$MFLOPS = \frac{30.000.000}{0{,}047360678 \cdot 10^6} = 633{,}436877$$

Obtenemos resultados diferentes respecto a la ejecución con sólo 10 componentes del vector.

código ensamblador generado de la parte de la suma de vectores

```
                call        clock_gettime
                xorl        %eax, %eax
                .p2align 4,,10
                .p2align 3
.L5:
                movsd       v1(%rax), %xmm0
                addq        $8, %rax
                addsd       v2-8(%rax), %xmm0
                movsd       %xmm0, v3-8(%rax)
                cmpq        %rbx, %rax
                jne         .L5
.L6:
                leaq        16(%rsp), %rsi
                xorl        %edi, %edi
                call        clock_gettime
```

2. Implementar un programa en C con OpenMP, a partir del código del Listado 1, que calcule en paralelo la suma de dos vectores (v3 = v1 + v2; v3(i)=v1(i)+v2(i), i=0,...N-1) usando las directivas `parallel` y `for`. Se debe paralelizar también las tareas asociadas a la inicialización de los vectores. Como en el código del Listado 1 se debe obtener el tiempo (*elapsed time*) que supone el cálculo de la suma. Para obtener este tiempo usar la función `omp_get_wtime()`, que proporciona el estándar OpenMP, en lugar de `clock_gettime()`. NOTAS: (1) el número de componentes N de los vectores debe ser un argumento de entrada al programa; (2) se deben inicializar los vectores antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para varios 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 suma los vectores y, al menos, el primer y último componente de v1, v2 y v3 (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

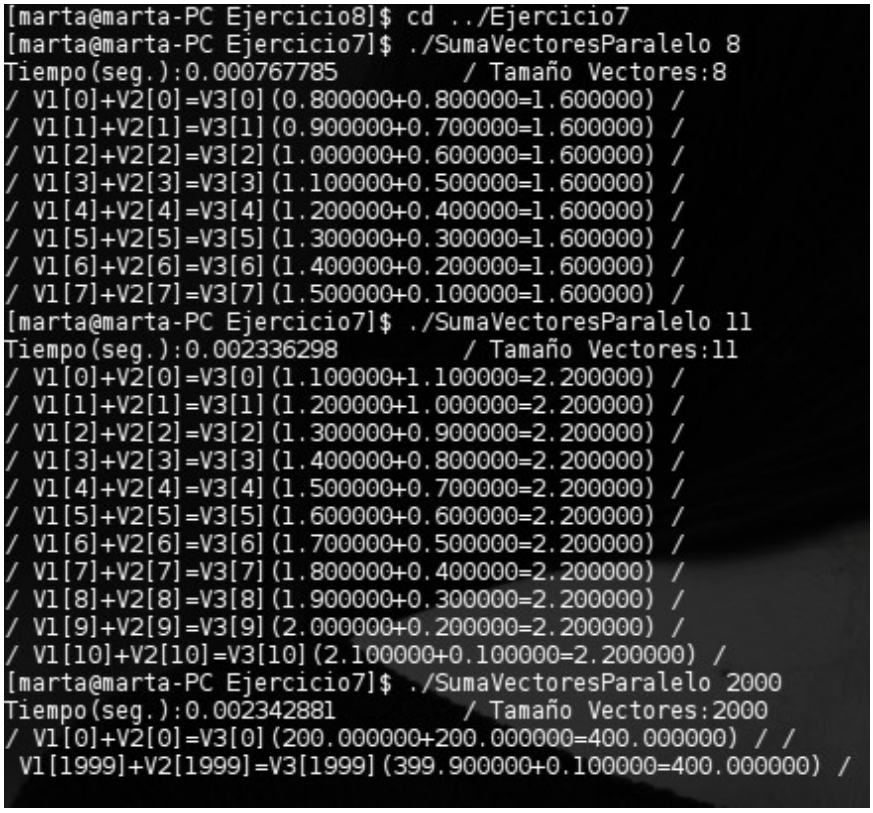
RESPUESTA: código fuente implementado

```
//Inicializar vectores
#pragma omp parallel for
    for(i=0; i<N; i++) {
        v1[i] = N*0.1+i*0.1;
        v2[i] = N*0.1-i*0.1; //los valores dependen de N
    }
double start = omp_get_wtime();
//Calcular suma de vectores
#pragma omp parallel for
```

```
    for(i=0; i<N; i++)
        v3[i] = v1[i] + v2[i];
double end = omp_get_wtime();
```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

CAPTURAS DE PANTALLA:



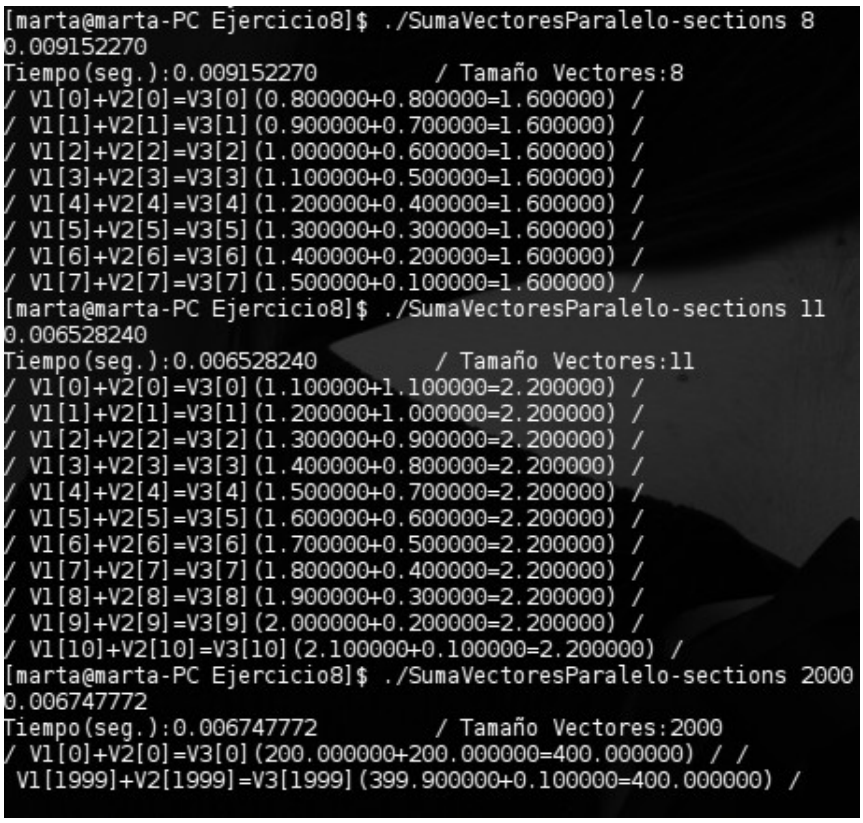
. Implementar un programa en C con OpenMP, a partir del código del Listado 1, que calcule en paralelo la suma de dos vectores usando las `parallel` y `sections/section` (se debe aprovechar el paralelismo de datos usando estas directivas en lugar de la directiva `for`); es decir, hay que repartir el trabajo (tareas) entre varios threads usando `sections/section`. Se debe paralelizar también las tareas asociadas a la inicialización de los vectores. Para obtener este tiempo usar la función `omp_get_wtime()` en lugar de `clock_gettime()`. NOTAS: (1) el número de componentes N de los vectores debe ser un argumento de entrada al programa; (2) se deben inicializar los vectores 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 suma los vectores y, al menos, el primer y último componente de v1, v2 y v3 (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

RESPUESTA: código fuente implementado

```
//Inicializar vectores
#pragma omp parallel sections
{
    #pragma omp section
    {
        for (primer_trozo=0; primer_trozo<(N/4); primer_trozo++) {
            v1[primer_trozo] = N*0.1+primer_trozo*0.1;
            v2[primer_trozo] = N*0.1-primer_trozo*0.1;
        }
    }
    #pragma omp section
    {
        for (segundo_trozo=(N/4); segundo_trozo<(N/2); segundo_trozo++) {
            v1[segundo_trozo] = N*0.1+segundo_trozo*0.1;
            v2[segundo_trozo] = N*0.1-segundo_trozo*0.1;
        }
    }
    #pragma omp section
    {
        for (tercer_trozo=(N/2); tercer_trozo<((3*N)/4); tercer_trozo++) {
            v1[tercer_trozo] = N*0.1+tercer_trozo*0.1;
            v2[tercer_trozo] = N*0.1-tercer_trozo*0.1;
        }
    }
    #pragma omp section
    {
        for (cuarto_trozo=((3*N)/4); cuarto_trozo<N; cuarto_trozo++) {
            v1[cuarto_trozo] = N*0.1+cuarto_trozo*0.1;
            v2[cuarto_trozo] = N*0.1-cuarto_trozo*0.1;
        }
    }
}
double start = omp_get_wtime();
//Calcular suma de vectores
#pragma omp parallel sections
{
    #pragma omp section
    {
        for (primer_trozo=0; primer_trozo<(N/4); primer_trozo++)
            v3[primer_trozo] = v1[primer_trozo] + v2[primer_trozo];
    }
    #pragma omp section
    {
        for (segundo_trozo=(N/4); segundo_trozo<(N/2); segundo_trozo++)
            v3[segundo_trozo] = v1[segundo_trozo] + v2[segundo_trozo];
    }
    #pragma omp section
    {
        for (tercer_trozo=(N/2); tercer_trozo<((3*N)/4); tercer_trozo++)
            v3[tercer_trozo] = v1[tercer_trozo] + v2[tercer_trozo];
    }
    #pragma omp section
    {
        for (cuarto_trozo=((3*N)/4); cuarto_trozo<N; cuarto_trozo++)
            v3[cuarto_trozo] = v1[cuarto_trozo] + v2[cuarto_trozo];
    }
}
double end = omp_get_wtime();
```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

CAPTURAS DE PANTALLA:



. ¿Cuántos threads y cuántos cores como máximo podría utilizar la versión que ha implementado en el ejercicio 7? Razone su respuesta. ¿Cuántos threads y cuantos cores como máximo podría utilizar la versión que ha implementado en el ejercicio 8? Razone su respuesta.

RESPUESTA: En el ejercicio 7, la directiva **for** crea tantas hebras como indique la constante **OMP_NUM_THREADS**, que en principio se corresponde con los cores del PC, pero se puede cambiar este valor. En cambio, en el ejercicio 8, nosotros hemos dividido el bucle en cuatro partes y por tanto, se crearán cuatro hebras.

. Rellenar una tabla como la Tabla 2 para atcgrid y otra para el PC local con los tiempos de ejecución de los programas paralelos implementados en los ejercicios 7 y 8 y el programa secuencial del Listado 1. Generar los ejecutables usando -O2. En la tabla debe aparecer el tiempo de ejecución del trozo de código que realiza la suma en paralelo (este es el tiempo que deben imprimir los programas). Ponga en la tabla el número de threads/cores que usan los códigos. Represente en una gráfica los tres tiempos.

RESPUESTA:

Tabla 2. Tiempos de ejecución en mi PC

Nº de Componentes	T. secuencial vect. Dinámicos 1 thread/core	T. paralelo (versión `for`) 12 threads/cores	T. paralelo (versión `sections`) 12 threads/cores
16384	0,000135216	0,000075820	0,000083318
32768	0,000232114	0,000088935	0,000089327
65536	0,000435476	0,006272339	0,000146079
131072	0,000638859	0,000316963	0,000311186
262144	0,001283828	0,000730708	0,000576730
524288	0,002065432	0,003428345	0,001396838
1048576	0,003045841	0,002725269	0,002591328
2097152	0,006066143	0,008345175	0,005439702
4194304	0,011648756	0,010454436	0,012273216
8388608	0,023282247	0,021032311	0,021301452
16777216	0,046915376	0,042721923	0,042935271
33554432	0,094922471	0,086561643	0,087815215
67108864	0,205208506	0,177843853	0,186301955

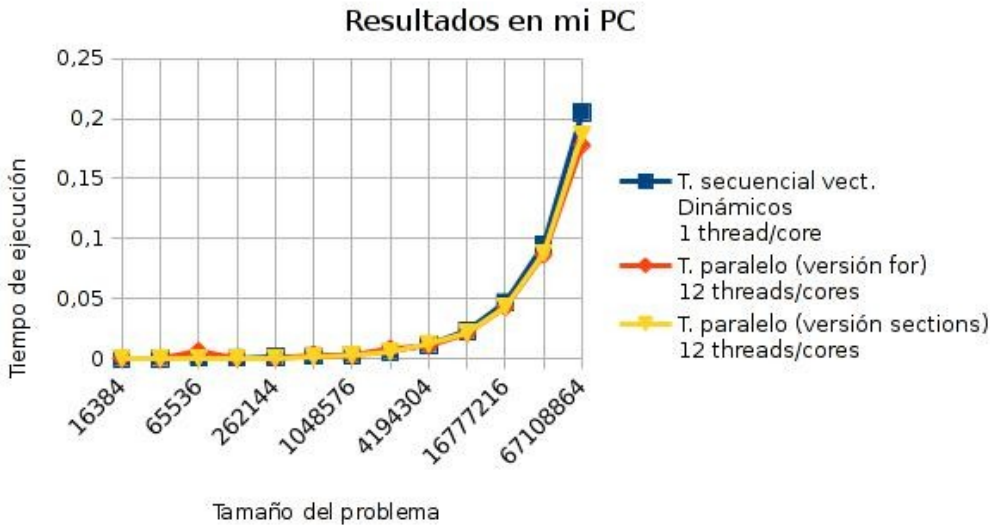
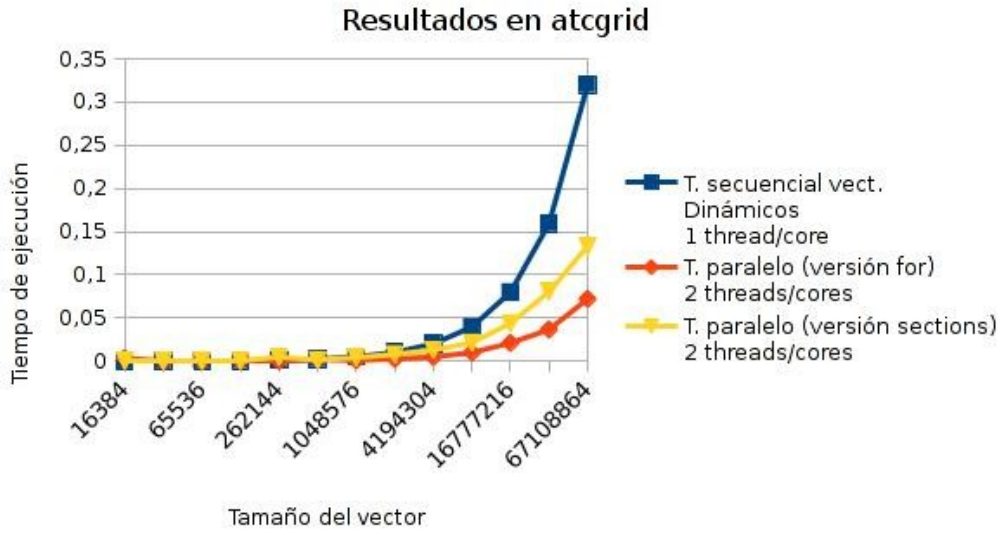


Tabla 2. Tiempos de ejecución en atcgrid

Nº de Componentes	T. secuencial vect. Dinámicos 1 thread/core	T. paralelo (versión `for`) 2 threads/cores	T. paralelo (versión `sections`) 2 threads/cores
16384	0,000069468	0,003376722	0,000071519
32768	0,000166689	0,000070985	0,000085421
65536	0,000331990	0,000124991	0,000196356
131072	0,000647124	0,000147114	0,000396098
262144	0,001304862	0,000295263	0,004788810
524288	0,002325230	0,002097613	0,001426470
1048576	0,005055141	0,000940561	0,004233311
2097152	0,010237943	0,002645325	0,007983003
4194304	0,020150268	0,005007029	0,013115505
8388608	0,039638230	0,009993202	0,021432165
16777216	0,079920444	0,021039304	0,043848220
33554432	0,159278615	0,036625341	0,080417794
67108864	0,320145535	0,072068237	0,133206408



. Rellenar una tabla como la Tabla 3 para el PC local con el tiempo de ejecución, tiempo de CPU del usuario y tiempo CPU del sistema obtenidos con `time` para el ejecutable del ejercicio 7 y para el programa secuencial del Listado 1. Ponga en la tabla el número de threads/cores que usan los códigos. ¿El tiempo de CPU que se obtiene es mayor o igual que el tiempo real (*elapsed*)? Justifique la respuesta.

RESPUESTA: El tiempo elapsed se corresponde con todo el tiempo de

ejecución del programa (incluyendo esperas de cualquier tipo), mientras que el tiempo de CPU es la suma del tiempo de ejecución en el espacio de usuario y en el espacio del kernel, por tanto, es improbable que el tiempo de cpu y el tiempo elapsed coincidan. Normalmente el elapsed suele ser mayor que el tiempo de CPU, pero como se ve en la tabla, hay casos en los que es al revés. Esto se debe a que cuando tenemos varios flujos de control, el elapsed será el flujo que más tarde en ejecutarse y el tiempo de CPU, la suma de los tiempo de CPU de todos los flujos de control.

Tabla 3. Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas.

Nº de Componentes	**Tiempo secuencial vect. Globales 1 thread/core**			**Tiempo paralelo/versión `for` 4 Threads/cores**		
	Elapsed	***CPU-user***	***CPU- sys***	***Elapsed***	***CPU-user***	***CPU-sys***
65536	0,004	0,000	0,000	0,008	0,000	0,020
131072	0,005	0,000	0,000	0,004	0,000	0,000
262144	0,008	0,000	0,000	0,016	0,010	0,030
524288	0,009	0,000	0,000	0,009	0,000	0,020
1048576	0,013	0,010	0,000	0,014	0,020	0,020
2097152	0,016	0,010	0,000	0,018	0,000	0,050
4194304	0,030	0,010	0,020	0,029	0,040	0,060
8388608	0,055	0,010	0,040	0,051	0,110	0,070
16777216	0,108	0,040	0,060	0,102	0,110	0,270
33554432	0,216	0,030	0,180	0,212	0,230	0,570
67108864	0,441	0,110	0,330	0,440	0,640	1,080