

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): Clara María Romero Lara

Grupo de prácticas y profesor de prácticas: D1

Fecha de entrega: 22-03-20

Fecha evaluación en clase: 25-03-20

---

Antes de comenzar a realizar el trabajo de este cuaderno consultar el fichero con los normas de prácticas que se encuentra en SWAD

---

#### Ejercicios basados en los ejemplos del seminario práctico

1. 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:** Captura que muestre el código fuente `bucle-forModificado.c`

```
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <omp.h>
6 int main(int argc, char **argv) {
7
8     int i, n = 9;
9
10    if(argc < 2) {
11        fprintf(stderr,"[ERROR] - Falta no iteraciones \n");
12        exit(-1);
13    }
14
15    n = atoi(argv[1]);
16
17    #pragma omp parallel for
18    for (i=0; i<n; i++)
19        printf("thread %d ejecuta la iteración %d del bucle\n",
20              omp_get_thread_num(),i);
21
22    return(0);
23 }
```

**RESPUESTA:** Captura que muestre el código fuente sectionsModificado.c

```

3 #include <stdio.h>
4 #include <omp.h>
5 void funcA() {
6     printf("En funcA: esta sección la ejecuta el thread %d\n",
7        omp_get_thread_num());
8 }
9 void funcB() {
10    printf("En funcB: esta sección la ejecuta el thread %d\n",
11       omp_get_thread_num());
12 }
13 int main() {
14     #pragma omp parallel sections
15     {
16         #pragma omp section
17             (void) funcA();
18
19         #pragma omp section
20             (void) funcB();
21     }
22
23     return 0;
24 }
```

2. Imprimir los resultados del programa single.c usando una directiva single dentro de la construcción parallel en lugar de imprimirlas 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:** Captura que muestre el código fuente singleModificado.c

```

6 int main() {
7
8     int n = 9, i, a, b[n];
9
10    for (i=0; i<n; i++)
11        b[i] = -1;
12
13    #pragma omp parallel
14    {
15        #pragma omp single
16        {
17            printf("Introduce valor de inicialización a: ");
18            scanf("%d", &a );
19            printf("Single ejecutada por el thread %d\n",
20               omp_get_thread_num());
21        }
22
23        #pragma omp for
24            for (i=0; i<n; i++)
25                b[i] = a;
26
27        #pragma omp single
28        {
29            for (i=0; i<n; i++) printf("b[%d] = %d\n",i,b[i]);
30            printf("\n");
31        }
32    }
33 }
```

### CAPTURAS DE PANTALLA:

```
~/bp1/ejer2 took 11s
+ gcc -fopenmp -O2 singleModificado.c -o singleMod

~/bp1/ejer2
+ ./singleMod
Introduce valor de inicialización a: 20
Single ejecutada por el thread 7
b[0] = 20      b[1] = 20      b[2] = 20      b[3] = 20      b[4] = 20      b[5] = 20      b[6] = 20      b[7] = 20      b[8]
```

3. 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:** Captura que muestre el código fuente `singleModificado2.c`

```
3 #include <stdio.h>
4 #include <omp.h>
5 void funcA() {
6     printf("En funcA: esta sección la ejecuta el thread %d\n",
7         omp_get_thread_num());
8 }
9 void funcB() {
10    printf("En funcB: esta sección la ejecuta el thread %d\n",
11        omp_get_thread_num());
12 }
13 int main() {
14     #pragma omp parallel sections
15     {
16         #pragma omp section
17             (void) funcA();
18
19         #pragma omp section
20             (void) funcB();
21     }
22
23     return 0;
24 }
```

### CAPTURAS DE PANTALLA:

```
~/bp1/ejer3
+ ./singleMod2
Introduce valor de inicialización a: 20
Single ejecutada por el thread 3
Depués de la región parallel imprime hebra:0
b[0] = 20
b[1] = 20
b[2] = 20
b[3] = 20
b[4] = 20
b[5] = 20
b[6] = 20
b[7] = 20
b[8] = 20
```

```
~/bp1/ejer3
+ ./singleMod2
Introduce valor de inicialización a: 20
Single ejecutada por el thread 2
Depués de la región parallel imprime hebra:0
b[0] = 20
b[1] = 20
b[2] = 20
b[3] = 20
b[4] = 20
b[5] = 20
b[6] = 20
b[7] = 20
b[8] = 20
```

```
~/bp1/ejer3 took 2s
+ ./singleMod2
Introduce valor de inicialización a: 20
Single ejecutada por el thread 4
Depués de la región parallel imprime hebra:0
b[0] = 20
b[1] = 20
b[2] = 20
b[3] = 20
b[4] = 20
b[5] = 20
b[6] = 20
b[7] = 20
b[8] = 20
```

```
~/bp1/ejer2
+ ./singleMod
Introduce valor de inicialización a: 20
Single ejecutada por el thread 4
b[0] = 20      b[1] = 20      b[2] = 20      b[3] = 20      b[4] = 20      b[5] = 20      b[6] = 20      b[7] = 20      b[8]
```

```
~/bp1/ejer2 took 2s
+ ./singleMod
Introduce valor de inicialización a: 20
Single ejecutada por el thread 7
b[0] = 20      b[1] = 20      b[2] = 20      b[3] = 20      b[4] = 20      b[5] = 20      b[6] = 20      b[7] = 20      b[8]
```

```
~/bp1/ejer2 took 2s
+ ./singleMod
Introduce valor de inicialización a: 20
Single ejecutada por el thread 3
b[0] = 20      b[1] = 20      b[2] = 20      b[3] = 20      b[4] = 20      b[5] = 20      b[6] = 20      b[7] = 20      b[8]
```

#### **RESPUESTA A LA PREGUNTA:**

Cuando usamos master, la hebra utilizada siempre es la 0. Cuando vamos con single, toma una hebra cualquiera.

4. ¿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:**

Si no hay directiva barrier, no se espera a que las hebras terminen los cálculos así que se pueden producir interfoliaciones en las que se imprime el resultado aunque no se haya terminado el cálculo.

---

## Resto de ejercicios

5. 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 atcgrid, 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 menor, mayor o igual que el tiempo real (*elapsed*)? Justifique la respuesta.

```
[ClaraRomeroLara d1estudiante23@atcgrid:~/bp1/ejer5] 2020-03-13 viernes
$ sbatch -p ac --wrap "time ./suma 10000000"
Submitted batch job 20548
[ClaraRomeroLara d1estudiante23@atcgrid:~/bp1/ejer5] 2020-03-13 viernes
$ ls
slurm-20548.out  suma  SumaVectores.c
[ClaraRomeroLara d1estudiante23@atcgrid:~/bp1/ejer5] 2020-03-13 viernes
$ cat slurm-20548.out
Tamaño Vectores:10000000 (4 B)
Tiempo:0.042077515      / Tamaño Vectores:10000000      / V1[0]+V2[0]=V3[0] (10000000.000000+10000
0.000000=2000000.000000) / / V1[9999999]+V2[9999999]=V3[9999999] (1999999.900000+0.100000=200000
0.000000) /

real    0m0.123s
user    0m0.057s
sys     0m0.054s
[ClaraRomeroLara d1estudiante23@atcgrid:~/bp1/ejer5] 2020-03-13 viernes
$
```

#### **CAPTURAS DE PANTALLA:**

User+Sys) 0.57+0.54 = 1.11

Real) = 1.23

El tiempo real es ligeramente mayor. El tiempo real es un cronómetro de lo que tarda algo en ejecutarse en total, mientras que el tiempo CPU (user+sys) es el tiempo que tarda en ejecutar las órdenes en modo usuario o modo kernel. El tiempo real puede ser mayor que el tiempo en CPU porque procesos como por ejemplo E/S es tiempo durante el cual el programa se está ejecutando (cronómetro, tiempo real) pero no es realmente ninguna instrucción en ejecución, así que no se cuenta en el tiempo en CPU.

6. 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). Razonar cómo se han obtenido los valores que se necesitan para calcular los MIPS y MFLOPS. Incorpore **el código ensamblador de la parte de la suma de vectores** en el cuaderno.

**CAPTURAS DE PANTALLA** (que muestren la generación del código ensamblador y del código ejecutable, y la obtención de los tiempos de ejecución):

```
~/bp1/ejer6
+ gcc -fopenmp -O2 SumaVectores.c -S

~/bp1/ejer6
+ gcc -fopenmp -O2 SumaVectores.c -o suma

~/bp1/ejer6
+ time ./suma 10000000
Tamaño Vectores:10000000 (4 B)
Tiempo:0.063274869          / 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) /
./suma 10000000  0.04s user 0.16s system 98% cpu 0.207 total

~/bp1/ejer6
+ time ./suma 10
Tamaño Vectores:10 (4 B)
Tiempo:0.000003668          / Tamaño Vectores:10      / V1[0]+V2[0]=V3[0](1.000000+1.000000=2.000000) // V1[9]+V2[9]=V3[9](1.900000+0.100000=2.000000) /
./suma 10  0.00s user 0.00s system 86% cpu 0.002 total
```

**RESPUESTA:** cálculo de los MIPS y los MFLOPS

- Teniendo en cuenta que mi ordenador tiene 1.60GHz (F), sabemos que puede calcular como máximo 1.6 instrucciones por ciclo (IPC)
- CPI = 1/IPC; CPI = 0.625
- MIPS = F/CPI\*10<sup>9</sup>; MIPS = 1.6\*10<sup>9</sup>/0.625\*10<sup>9</sup>; GIPS = 2.56**
- El número de operaciones en coma flotante es 4 (movsd, addsd, movsd, compl)
- GFLOPS = (nº operaciones en coma flotante \* iteraciones) / T
- 10000000 componentes) **GFLOPS = 4\*10000000 / 0.063274869 = 632162509.889984916**
- 10 componentes) **GFLOPS = 4\*10 / 0.000003668 = 109051254.089422028**

**RESPUESTA:** Captura que muesre el código ensamblador generado de la parte de la suma de vectores

```
call  clock_gettime@PLT
xorl  %eax, %eax
.p2align 4,,10
.p2align 3
.L6:
    movsd (%r12,%rax,8), %xmm0
    addsd 0(%r13,%rax,8), %xmm0
    movsd %xmm0, (%r14,%rax,8)
    addq   $1, %rax
    cmpl  %eax, %ebp
    ja   .L6
    leaq   16(%rsp), %rsi
    xorl  %edi, %edi
    call  clock_gettime@PLT
```

7. Implementar un programa en C con OpenMP, a partir del código del Listado 1, que calcule en paralelo la suma de dos vectores ( $v3 = v1 + v2$ ;  $v3(i) = v1(i) + v2(i)$ ,  $i=0, \dots, N-1$ ) 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:** Captura que muestre el código fuente implementado

```

57 //Inicializar vectores
58 for(i=0; i<N; i++){
59     v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
60 }
61
62 double tiempo_ini = omp_get_wtime();
63
64 //Calcular suma de vectores
65 #pragma omp parallel for
66 for(i=0; i<N; i++)
67     v3[i] = v1[i] + v2[i];
68
69 double tiempo_fin = omp_get_wtime();
70 double tiempo_total = tiempo_fin - tiempo_ini;
71
72
73 //Imprimir resultado de la suma y el tiempo de ejecución
74 if (N<10) {
75     printf("Tiempo:%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
76     for(i=0; i<N; i++)
77         printf("/ V1[%d]+V2[%d]=V3[%d] (%8.6f+%8.6f=%8.6f) /\n",
78             i,i,i,v1[i],v2[i],v3[i]);
79 }
80 else
81     printf("Tiempo:%11.9f\t / Tamaño Vectores:%u\t/ V1[0]+V2[0]=V3[0]-
82 (%8.6f+%8.6f=%8.6f) /\t V1[%d]+V2[%d]=V3[%d] (%8.6f+%8.6f=%8.6f) /\n",
83             ncgt,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
84
85 #ifdef VECTOR_DYNAMIC
86 free(v1); // libera el espacio reservado para v1
87 free(v2); // libera el espacio reservado para v2
88 free(v3); // libera el espacio reservado para v3
89#endif
90 return 0;

```

**(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)**

**CAPTURAS DE PANTALLA (compilación y ejecución para N=8 y N=11):**

```

~/bp1/ejer7
+ gcc -O2 -lgomp SumaVectoresModificado.c -o SumaVectoresMod -lrt
~/bp1/ejer7
+ ls
SumaVectoresMod  SumaVectoresModificado.c

~/bp1/ejer7
+ ./SumaVectoresMod 11
Tamaño Vectores:11 (4 B)
Tiempo:0.0000000000 / Tamaño Vectores:11 / V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000) / / V1[10]+V2[10]=V3[10](2.100000+0.100000=2.200000) /

~/bp1/ejer7
+ ./SumaVectoresMod 8
Tamaño Vectores:8 (4 B)
Tiempo:0.0000000000 / Tamaño Vectores:8 /
/ V1[0]+V2[0]=V3[0](0.800000+0.800000=1.600000) /
/ V1[1]+V2[1]=V3[1](0.900000+0.700000=1.600000) /
/ V1[2]+V2[2]=V3[2](1.000000+0.600000=1.600000) /
/ V1[3]+V2[3]=V3[3](1.100000+0.500000=1.600000) /
/ V1[4]+V2[4]=V3[4](1.200000+0.400000=1.600000) /
/ V1[5]+V2[5]=V3[5](1.300000+0.300000=1.600000) /
/ V1[6]+V2[6]=V3[6](1.400000+0.200000=1.600000) /
/ V1[7]+V2[7]=V3[7](1.500000+0.100000=1.600000) /

```

8. 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 parallelizar 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); (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:** Captura que muestre el código fuente implementado

```

60     double tiempo_ini, tiempo_fin, tiempo_total;
61
62 #pragma omp parallel
63 {
64     //Inicializar vectores
65     #pragma omp sections
66     {
67         #pragma omp section
68         for(i=0; i<N; i++){
69             v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
70         }
71
72         #pragma omp section
73         for(i=0; i<N*0.75; i++){
74             v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
75         }
76
77         #pragma omp section
78         for(i=0; i<N*0.5; i++){
79             v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
80         }
81
82         #pragma omp section
83         for(i=0; i<N*0.25; i++){
84             v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
85         }
86     }
87
88     #pragma omp single
89     tiempo_ini = omp_get_wtime();
90
91
92     //Calcular suma de vectores
93     #pragma omp sections
94     {
95         #pragma omp section
96         for(i=0; i<N; i++)
97             v3[i] = v1[i] + v2[i];
98
99         #pragma omp section
100        for(i=0; i<N*0.75; i++)
101            v3[i] = v1[i] + v2[i];
102
103        #pragma omp section
104        for(i=0; i<N*0.5; i++)
105            v3[i] = v1[i] + v2[i];
106
107        #pragma omp section
108        for(i=0; i<N*0.25; i++)
109            v3[i] = v1[i] + v2[i];
110    }
111
112     #pragma omp single
113     tiempo_fin = omp_get_wtime();
114
115 }
116
117 tiempo_total = tiempo_fin - tiempo_ini;

```

**(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)**

**CAPTURAS DE PANTALLA (compilación y ejecución para N=8 y N=11):**

```

~/bp1/ejer8
+ gcc -O2 -fopenmp SumaVectoresModificado2.c -o SumaVectoresMod2 -lrt

~/bp1/ejer8
+ ./SumaVectoresMod2 8
Tamaño Vectores:8 (4 B)
Tiempo:0.000000000 / Tamaño Vectores:8
/ V1[0]+V2[0]=V3[0](0.800000+0.800000=1.600000) /
/ V1[1]+V2[1]=V3[1](0.900000+0.700000=1.600000) /
/ V1[2]+V2[2]=V3[2](1.000000+0.600000=1.600000) /
/ V1[3]+V2[3]=V3[3](1.100000+0.500000=1.600000) /
/ V1[4]+V2[4]=V3[4](1.200000+0.400000=1.600000) /
/ V1[5]+V2[5]=V3[5](1.300000+0.300000=1.600000) /
/ V1[6]+V2[6]=V3[6](1.400000+0.200000=1.600000) /
/ V1[7]+V2[7]=V3[7](1.500000+0.100000=1.600000) /

~/bp1/ejer8
+ ./SumaVectoresMod2 11
Tamaño Vectores:11 (4 B)
Tiempo:0.000000000 / Tamaño Vectores:11 / V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000) / / V1[10]+V2[10]=V3[10](2.100000+0.100000=2.200000) /

```

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

**RESPUESTA:**

**Ambos ejercicios pueden hacer uso de todos los threads y cores disponibles en la máquina que los ejecuta (en mi caso, 8 cores lógicos, 4 cores físicos). Esto se debe a que no hemos definido en ningún momento la variable OMP\_NUM\_THREADS.**

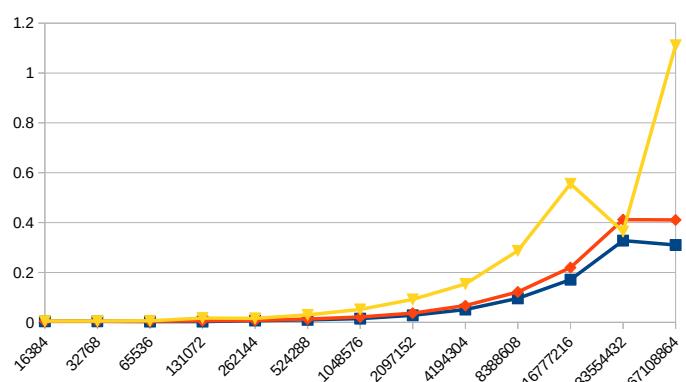
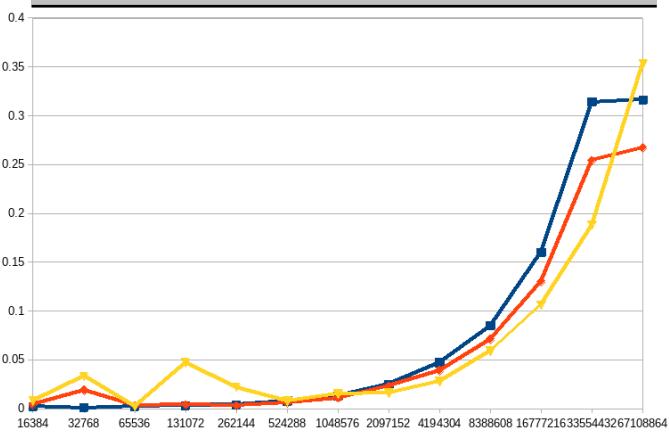
10. Rellenar una tabla como la Tabla 2 para atcgrid y otra para su PC con los tiempos de ejecución de los programas paralelos implementados en los ejercicios 7 y 8 y el programa secuencial del Listado 1. Generar los ejecutables usando -O2. En la tabla debe aparecer el tiempo de ejecución del trozo de 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 (use el máximo número de cores físicos del computador que como máximo puede aprovechar el código, no use un número de threads superior al número de cores físicos). Represente en una gráfica los tres tiempos. NOTA: Nunca ejecute código que imprima todos los componentes del resultado cuando este número sea elevado.

**RESPUESTA:**

**Tabla 2.** Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas. Sustituir en el encabezado de la tabla “?” por el número de threads utilizados, que debe coincidir con el número de cores físicos del computador que como máximo puede aprovechar el código.

Nº de Componentes	T. secuencial vect. Globales 1 thread/core	T. paralelo (versión for) 8 threads/ 4 cores	T. paralelo (versión sections) 8 threads/ 4 cores
<b>16384</b>	0.002	0.004	0.008
<b>32768</b>	0.001	0.019	0.033
<b>65536</b>	0.002	0.003	0.002
<b>131072</b>	0.003	0.004	0.047
<b>262144</b>	0.004	0.003	0.022
<b>524288</b>	0.008	0.006	0.008
<b>1048576</b>	0.013	0.011	0.015
<b>2097152</b>	0.025	0.023	0.016
<b>4194304</b>	0.047	0.039	0.028
<b>8388608</b>	0.085	0.071	0.059
<b>16777216</b>	0.160	0.130	0.106
<b>33554432</b>	0.314	0.254	0.188
<b>67108864</b>	0.316	0.267	0.353

Nº de Componentes	T. secuencial vect. Globales 1 thread/core	T. paralelo (versión for) 24 threads/12 cores	T. paralelo (versión sections) 24 threads/ 12cores
<b>16384</b>	0.004	0.004	0.005
<b>32768</b>	0.004	0.004	0.005
<b>65536</b>	0.003	0.004	0.006
<b>131072</b>	0.003	0.006	0.017
<b>262144</b>	0.007	0.008	0.016
<b>524288</b>	0.010	0.013	0.030
<b>1048576</b>	0.015	0.021	0.052
<b>2097152</b>	0.028	0.037	0.092
<b>4194304</b>	0.051	0.067	0.154
<b>8388608</b>	0.096	0.122	0.287
<b>16777216</b>	0.171	0.220	0.556
<b>33554432</b>	0.328	0.412	0.364
<b>67108864</b>	0.310	0.411	1.111



11. Rellenar una tabla como la Tabla 3 para atcgrid 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:**

**Tabla 3.** Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas. Sustituir en el encabezado de la tabla “*?*” por el número de threads utilizados.

Nº de Componente s	Tiempo secuencial vect. Globales			Tiempo paralelo/versión for 24 Threads/ 12cores		
	Elapsed	1 thread/core CPU-user	CPU- sys	Elapsed	CPU-user	CPU- sys
65536	0.046	0	0.003	0.061	0.001	0.004
131072	0.055	0	0.003	0.021	0	0.003
262144	0.055	0.003	0.002	0.039	0.002	0.002
524288	0.092	0.002	0.004	0.050	0.004	0.004
1048576	0.086	0.005	0.004	0.099	0.005	0.006
2097152	0.107	0.006	0.010	0.326	0.011	0.009
4194304	0.083	0.012	0.015	0.285	0.019	0.018
8388608	0.100	0.024	0.028	0.115	0.070	0.056
16777216	0.115	0.056	0.039	0.187	0.141	0.085
33554432	0.0193	0.096	0.076	0.320	0.252	0.169
67108864	0.348	0.167	0.161	0.365	0.263	0.161