

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): Francisco Rodríguez Jiménez

Grupo de prácticas y profesor de prácticas: C1 Juan José Escobar

Fecha de entrega:

Fecha evaluación en clase:

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`

```
Abrir [icono] *bucle-forModificado.c
~/Documentos/3º CURSO/2 Cuatrimestre/AC...grupo reducido_/6. PRACTICAS/bp1/ejer1 Guardar [icono]

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

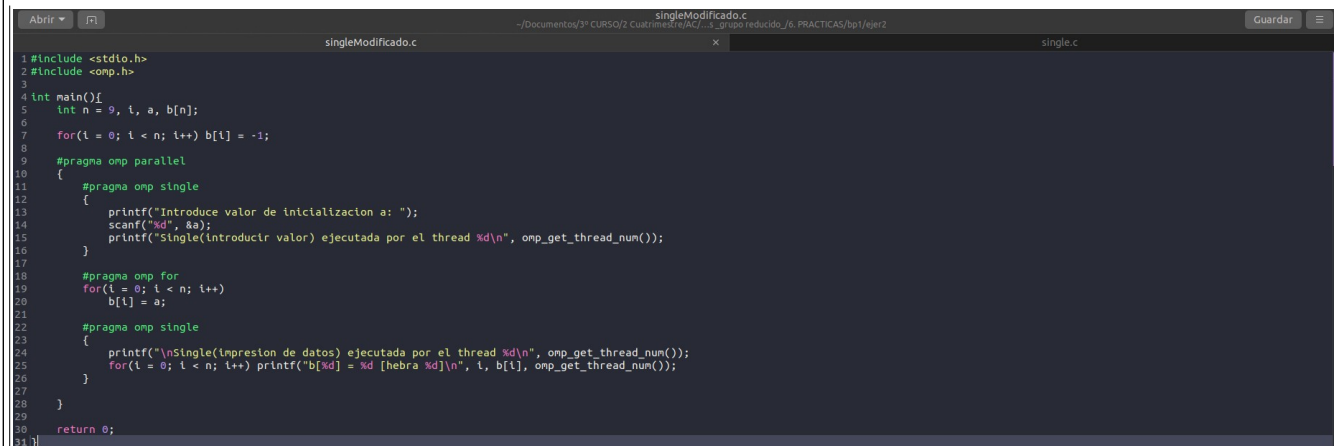
RESPUESTA: Captura que muestre el código fuente `sectionsModificado.c`

```
Abrir [icono] sectionsModificado.c
~/Documentos/3º CURSO/2 Cuatrimestre/AC...grupo reducido_/6. PRACTICAS/bp1/ejer1 Guardar [icono]

1 #include <stdio.h>
2 #include <omp.h>
3
4 void funcA(){
5     printf("En FuncA: esta seccion la ejecuta el thread %d\n", omp_get_thread_num());
6 }
7
8 void funcB(){
9     printf("En FuncB: esta seccion la ejecuta el thread %d\n", omp_get_thread_num());
10 }
11
12
13 int main(){
14
15     #pragma omp parallel sections
16     {
17         #pragma omp section
18         (void)funcA();
19         #pragma omp section
20         (void)funcB();
21     }
22
23     return 0;
24 }
25
```

2. 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: Captura que muestre el código fuente `singleModificado.c`

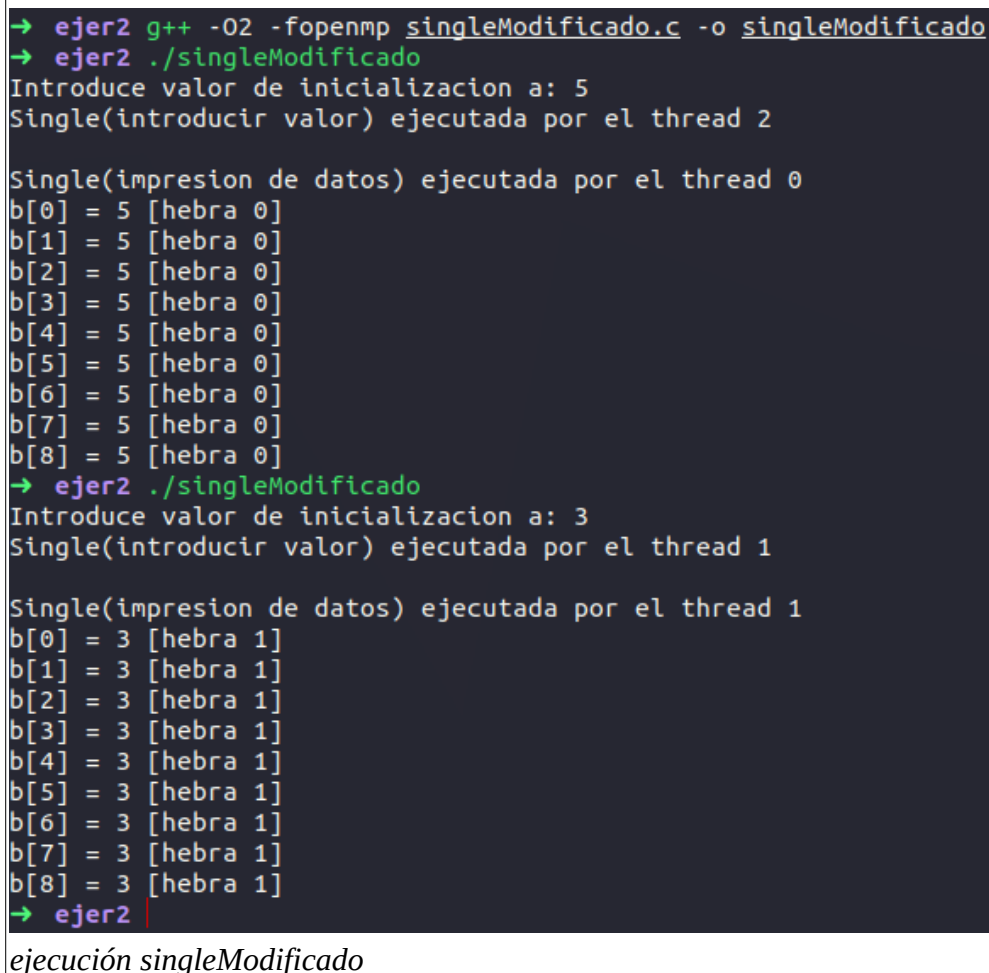


```

1 #include <stdio.h>
2 #include <omp.h>
3
4 int main(){
5     int n = 9, i, a, b[n];
6
7     for(i = 0; i < n; i++) b[i] = -1;
8
9     #pragma omp parallel
10    {
11        #pragma omp single
12        {
13            printf("Introduce valor de inicializacion a: ");
14            scanf("%d", &a);
15            printf("Single(introducir valor) ejecutada por el thread %d\n", omp_get_thread_num());
16        }
17
18        #pragma omp for
19        for(i = 0; i < n; i++)
20            b[i] = a;
21
22        #pragma omp single
23        {
24            printf("\nSingle(impression de datos) ejecutada por el thread %d\n", omp_get_thread_num());
25            for(i = 0; i < n; i++) printf("b[%d] = %d [hebra %d]\n", i, b[i], omp_get_thread_num());
26        }
27    }
28
29    return 0;
30 }

```

CAPTURAS DE PANTALLA:



```

→ ejer2 g++ -O2 -fopenmp singleModificado.c -o singleModificado
→ ejer2 ./singleModificado
Introduce valor de inicializacion a: 5
Single(introducir valor) ejecutada por el thread 2

Single(impression de datos) ejecutada por el thread 0
b[0] = 5 [hebra 0]
b[1] = 5 [hebra 0]
b[2] = 5 [hebra 0]
b[3] = 5 [hebra 0]
b[4] = 5 [hebra 0]
b[5] = 5 [hebra 0]
b[6] = 5 [hebra 0]
b[7] = 5 [hebra 0]
b[8] = 5 [hebra 0]
→ ejer2 ./singleModificado
Introduce valor de inicializacion a: 3
Single(introducir valor) ejecutada por el thread 1

Single(impression de datos) ejecutada por el thread 1
b[0] = 3 [hebra 1]
b[1] = 3 [hebra 1]
b[2] = 3 [hebra 1]
b[3] = 3 [hebra 1]
b[4] = 3 [hebra 1]
b[5] = 3 [hebra 1]
b[6] = 3 [hebra 1]
b[7] = 3 [hebra 1]
b[8] = 3 [hebra 1]
→ ejer2

```

ejecución singleModificado

Como se puede comprobar la impresión de los resultados es ejecutada simplemente a una hebra, debido a la directiva **#pragma omp single**, se puede observar que una sola hebra imprime el bucle for completamente.

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`



```

1 #include <stdio.h>
2 #include <omp.h>
3
4 int main() {
5     int n = 9, i, a, b[n];
6
7     for(i = 0; i < n; i++) b[i] = -1;
8
9     #pragma omp parallel
10    {
11        #pragma omp single
12        {
13            printf("Introduce valor de inicializacion a: ");
14            scanf("%d", &a);
15            printf("Single(introducir valor) ejecutada por el thread %d\n", omp_get_thread_num());
16        }
17
18        #pragma omp for
19        for(i = 0; i < n; i++)
20            b[i] = a;
21
22        #pragma omp master
23        {
24            printf("\nSingle(impression de datos) ejecutada por el thread %d\n", omp_get_thread_num());
25            for(i = 0; i < n; i++) printf("b[%d] = %d [hebra %d]\n", i, b[i], omp_get_thread_num());
26        }
27    }
28
29    return 0;
30 }

```

CAPTURAS DE PANTALLA:

```

Archivo  Editar  Ver  Buscar  Terminal  Ayuda
→ ejer3 g++ -O2 -fopenmp singleModificado2.c -o singleModificado2
→ ejer3 ./singleModificado2
Introduce valor de inicializacion a: 10
Single(introducir valor) ejecutada por el thread 2

Single(impression de datos) ejecutada por el thread 0
b[0] = 10 [hebra 0]
b[1] = 10 [hebra 0]
b[2] = 10 [hebra 0]
b[3] = 10 [hebra 0]
b[4] = 10 [hebra 0]
b[5] = 10 [hebra 0]
b[6] = 10 [hebra 0]
b[7] = 10 [hebra 0]
b[8] = 10 [hebra 0]
→ ejer3 ./singleModificado2
Introduce valor de inicializacion a: 5
Single(introducir valor) ejecutada por el thread 1

Single(impression de datos) ejecutada por el thread 0
b[0] = 5 [hebra 0]
b[1] = 5 [hebra 0]
b[2] = 5 [hebra 0]
b[3] = 5 [hebra 0]
b[4] = 5 [hebra 0]
b[5] = 5 [hebra 0]
b[6] = 5 [hebra 0]
b[7] = 5 [hebra 0]
b[8] = 5 [hebra 0]
→ ejer3 |

```

ejecución singleModificado2

RESPUESTA A LA PREGUNTA:

Que la impresión de datos es impresa por la hebra 0 la cual es la master (principal) debido a la directiva **#pragma omp master**, el problema de esta directiva es que no tiene barreras implícitas, como la **#pragma omp single**, y el resultado no sera siempre el correcto.

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:

Esto es debido a que la directiva **master no** contiene **barreras implícitas**, por lo cual si no se pone una barrera antes de ser ejecutada la directiva master (la cual calcula la suma), puede ser realizada sin que las demás hebras hayan terminado el trabajo anterior. Con lo cual puede dar resultados incorrectos.

Lo que hace la directiva **barrier** es esperar a que terminen todas la hebras, una vez que terminan continua.

Resto de ejercicios

5. El programa secuencial C del Listado 1 calcula la suma de dos vectores ($v3 = v1 + v2$; $v3(i) = v1(i) + v2(i)$, $i=0, \dots, N-1$). 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.

CAPTURAS DE PANTALLA:

```

[FranciscoRodriguezJlmenez] Diestudiante20 @ ~/bp1/ejer5
18:12:35 $ g++ -O2 -fopenmp SumaVectoresC.c -o SumaVectoresC
[FranciscoRodriguezJlmenez] Diestudiante20 @ ~/bp1/ejer5
18:12:39 $ echo 'time $HOME/bp1/ejer5/SumaVectoresC 10000000' | qsub -q ac
12824.atcgrid
[FranciscoRodriguezJlmenez] Diestudiante20 @ ~/bp1/ejer5
18:12:45 $ cat STDIN.e12824
Tamaño Vectores:10000000 (4 B)
Tiempo:0.041806312 / Tamaño Vectores:10000000 / V1[0]+V2[0]=V3[0](1.000000.000000+1.000000.000000=2.000000.000000) // V1[999999]+V2[999999]=V3[999999](1.900000+0.100000=2.000000) /
[FranciscoRodriguezJlmenez] Diestudiante20 @ ~/bp1/ejer5
18:12:49 $ cat STDIN.e12824
real    0m0.110s
user    0m0.058s
sys     0m0.050s
[FranciscoRodriguezJlmenez] Diestudiante20 @ ~/bp1/ejer5
18:12:53 $

```

orden time en SumaVectoresGlobales

- **real** 0m0.110s
- **user** 0m0.058s
- **sys** 0m0.050s

$$\text{sys} + \text{user} = 0.108\text{s} < \text{real}$$

El término «**tiempo real**» en este contexto se refiere al tiempo transcurrido, como si fuese un cronómetro. El tiempo de CPU total (tiempo de usuario + tiempo de sistema) puede ser mayor o menor que ese valor. Puesto que un programa puede pasar algún tiempo de espera y no ejecutar nada (ya sea en modo usuario o modo sistema) el tiempo real puede ser mayor que el tiempo total de CPU. Puesto que un programa puede **bifurcar hijos** cuyo tiempo de CPU (tanto de usuario como de sistema) es sumado a los valores mostrados por el comando `time`, el tiempo de CPU total puede ser mayor que el tiempo real.

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):

```

kazz@kazzubuntu:~/Documentos/3º CURSO/2 Cuatrimestre/AC/Practicas_grupo reducido/J6. PRACTICAS/bp1/ejer6
→ ejer6 gcc -O2 -fopenmp SumaVectoresC.c -S SumaVectoresC.S && echo "Codigo ensamblador generado"; gcc -O2 -fopenmp SumaVectoresC.c -o SumaVectoresC && echo "Codigo ejecutable generado"
Codigo ensamblador generado
Codigo ejecutable generado
→ ejer6 cat SumaVectoresC.S | tail -10
.align 8
.LC2:
.long 2576980378
.long 1069120089
.align 8
.LC5:
.long 0
.long 1104006501
.ident "GCC: (Ubuntu 7.3.0-27ubuntu1-18.04) 7.3.0"
.section .note.GNU-stack,"",@progbits
→ ejer6 ./SumaVectoresC 10 ; ./SumaVectoresC 10000000
Tamaño Vectores:10 (4 B)
Tiempo:0.00001300 / 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) /
Tamaño Vectores:10000000 (4 B)
Tiempo:0.030747539 / Tamaño Vectores:10000000 / V1[0]+V2[0]=V3[0](1.000000.000000+1.000000.000000=2.000000.000000) // V1[999999]+V2[999999]=V3[999999](1.999999.900000+0.100000=2.000000.000000) /
→ ejer6

```

RESPUESTA: cálculo de los MIPS y los MFLOPS

- $MIPS = F / (CPI * 10^9)$
- $MFLOPS = \text{Operaciones en Coma Flotante} / (TCPU * 10^9)$

MIPS → Dado que mi PC tiene un procesador: Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
2.50GHz → tiene recursos para completar 2,5 instrucciones por ciclo como máximo (IPC = 2,5) → $CPI = 1/IPC = 0,4$.

- $MIPS = 2,5 * 10^9 / (0,4 * 10^9) = 6,25 \text{ GIPS}$

(11 instrucciones para suma de vectores) de las cuales $TCPU = 11 * 0,4 / (2,5 * 10^9) = 1,76E-9s$

Para 10 componentes = (5 instrucciones en coma flotante * 10 iteraciones) / $0,000001300 * 10^6 = 3MFLOPS$

Para 10000000 componentes = $(5 * 10000000) / 0,030747539 * 10^9 = 1,62GLOPS$

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

```

93:          call    clock_gettime@PLT
94:          xorl    %eax, %eax
95:          .p2align 4,,10
96:          .p2align 3
97: .L6:
98:          movsd   (%r12,%rax,8), %xmm0
99:          addsd   0(%r13,%rax,8), %xmm0
100:         movsd   %xmm0, (%r14,%rax,8)
101:         addq    $1, %rax
102:         cmpl    %eax, %ebp
103:         ja      .L6
104:         leaq    16(%rsp), %rsi
105:         xorl    %edi, %edi
106:         call    clock_gettime@PLT

```

bucle for(sumaVectores) en ensamblador, generado con la opción de optimización -O2

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

```

67 //Inicializar vectores
68 #pragma omp parallel for
69 for(i=0; i<N; i++){
70     v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
71 }
72
73
74 double tiempo1 = omp_get_wtime();
75
76 //Calcular suma de vectores
77 #pragma omp parallel for
78 for(i=0; i<N; i++){
79     v3[i] = v1[i] + v2[i];
80 }
81
82 double tiempo2 = omp_get_wtime();
83
84 double tiempo_total = tiempo2 - tiempo1;
85
86 //Imprimir resultado de la suma y el tiempo de ejecución
87 if (N<10) {
88     printf("Tiempo:%11.9f\t / Tamaño Vectores:%u\n",tiempo_total,N);
89     #pragma omp parallel for
90     for(i=0; i<N; i++){
91         printf("/ V1[%d]+V2[%d]=V3[%d](%.6f+%.6f=%.6f) /\n",
92             i,i,v1[i],v2[i],v3[i]);
93     }
94 }
95 else
96     printf("Tiempo:%11.9f\t / Tamaño Vectores:%u\t/ V1[0]+V2[0]=V3[0](%.6f+%.6f=%.6f) / / V1[%d]+V2[%d]=V3[%d](%.6f+%.6f=%.6f) /\n",
97         tiempo_total,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
98
99 #ifdef VECTOR_DYNAMIC
100 free(v1); // libera el espacio reservado para v1
101 free(v2); // libera el espacio reservado para v2
102 free(v3); // libera el espacio reservado para v3
103 #endif
104 return 0;
105 }

```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)**CAPTURAS DE PANTALLA (compilación y ejecución para N=8 y N=11):**

```

→ ejer7 gcc -O2 -fopenmp SumaVectoresC_parallel.c SumaVectoresC_parallel
→ ejer7 ./SumaVectoresC_parallel 8
Tamaño Vectores:8 (4 B)
Tiempo:0.000001619 / 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) /
→ ejer7 ./SumaVectoresC_parallel 11
Tamaño Vectores:11 (4 B)
Tiempo:0.000001860 / 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) /
→ ejer7

```

ejecución sumaVectores_parallel para N=8, N=11

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 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, v_3 , 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 v_1 , v_2 y v_3 (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

RESPUESTA: Captura que muestre el código fuente implementado

```

29 // inicializamos los vectores
30 double tiempo_inicial, tiempo_final, tiempo_total;
31 #pragma omp parallel
32 {
33     #pragma omp sections
34     {
35         #pragma omp section
36         for(i=0; i<N/4; i++){
37             v1[i] = N*0.1+i*0.1;
38             v2[i] = N*0.1-i*0.1;
39         }
40
41         #pragma omp section
42         for(j=N/4; j<N/2; j++){
43             v1[j] = N*0.1+j*0.1;
44             v2[j] = N*0.1-j*0.1;
45         }
46
47         #pragma omp section
48         for(k=N/2; k<3*N/4; k++){
49             v1[k] = N*0.1+k*0.1;
50             v2[k] = N*0.1-k*0.1;
51         }
52
53         #pragma omp section
54         for(l=3*N/4; l<N; l++){
55             v1[l] = N*0.1+l*0.1;
56             v2[l] = N*0.1-l*0.1;
57         }
58     }
59 }
60
61 #pragma omp single
62 tiempo_inicial = omp_get_wtime();
63
64 #pragma omp sections
65 {
66     #pragma omp section
67     for(i=0; i<N/4; i++){
68         v3[i] = v1[i] + v2[i];
69     }
70     #pragma omp section
71     for(j=N/4; j<N/2; j++){
72         v3[j] = v1[j] + v2[j];
73     }
74     #pragma omp section
75     for(k=N/2; k<3*N/4; k++){
76         v3[k] = v1[k] + v2[k];
77     }
78     #pragma omp section
79     for(l=3*N/4; l<N; l++){
80         v3[l] = v1[l] + v2[l];
81     }
82 }
83
84 #pragma omp single
85 tiempo_final = omp_get_wtime();
86
87 tiempo_total = tiempo_final - tiempo_inicial;
88
89 //Imprimir resultado de la suma y el tiempo de ejecución
90 if (N<10) {
91     printf("Tiempo:%11.9f\t / Tamaño Vectores:%u\n",tiempo_total,N);
92     for(i=0; i<N; i++){
93         printf("/ V1[%d]+V2[%d]=V3[%d] (%8.6f+%8.6f=%8.6f) /\n",
94             i,i,i,v1[i],v2[i],v3[i]);
95     }
96 }
97 else
98     printf("Tiempo:%11.9f\t / Tamaño Vectores:%u\t / V1[0]+V2[0]=V3[0] (%8.6f+%8.6f=%8.6f) / / V1[%d]+V2[%d]=V3[%d] (%8.6f+%8.6f=%8.6f) /\n",
99         tiempo_total,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);

```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)**CAPTURAS DE PANTALLA (compilación y ejecución para N=8 y N=11):**


```

cazz@cazz-ubuntu: ~/Documentos/3º CURSO/2 Cuatrimestre/AC/Practicas_grupo reducido_/6. PRACTICAS/bp1/ejer8
Archivo Editar Ver Buscar Terminal Ayuda
→ ejer8 gcc -O2 -fopenmp SumaVectoresC_parallelv2.c -o SumaVectoresC_parallelv2
→ ejer8 ./SumaVectoresC_parallelv2 8; ./SumaVectoresC_parallelv2 11
Tiempo:0.000001814 / 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) /
Tiempo:0.000001680 / 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) /
→ ejer8

```

ejecución sumaVectores_parallelv2 para N=8, N=11

9. ¿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, el número de threads y cores como máximo utilizados son los de mi procesador o sea 4, a menos que se establezca otro numero de hebras con la función `omp_set_num_threads(x)` el número de cores máximo sera siempre el de mi procesador, En el ejercicio 8 ocurre lo mismo.

También lo he averiguado gracias a las funciones `omp_get_max_threads()` que devuelve el máximo numero de hebras que puede haber en una región paralela, y `omp_get_num_procs()` devuelve el número de procesadores disponibles para el programa.

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:

Tiempos PC:

Nº de Componentes	T.secuencial vect. Globales 1 thread/core	T.paralelo (versión for) 4 threads/cores	T.paralelo (versión sections) 4 threads/cores
16384	5,1145E-05	5,0187E-05	1,703E-05
32768	0,000100776	8,9489E-05	3,6345E-05
65536	0,000201389	0,000185574	5,4407E-05
131072	0,000488229	0,000348467	0,000111602
262144	0,000948217	0,000732147	0,000310074
524288	0,001813372	0,001485515	0,001003447
1048576	0,003914696	0,002384022	0,002095468
2097152	0,006868779	0,012879619	0,003988419
4194304	0,012851695	0,018429159	0,008580042
8388608	0,025585418	0,029245818	0,014636444
16777216	0,05041743	0,0529045	0,029141357
33554432	0,101063928	0,068929909	0,057982105
67108864	0,208512269	0,134628913	0,115038586

Tiempos atcgrid:

Nº de Componentes	T.secuencial vect. Globales 1 thread/core	T.paralelo (versión for) 24 threads/cors	T.paralelo (versión sections) 24 threads/cors
16384	0,000440449	0,003096343	6,8509E-05
32768	0,00047875	0,003179811	0,00429079
65536	0,000361199	0,003298142	0,000211979
131072	0,000509669	0,003405731	0,002372193
262144	0,001250358	0,00390062	0,000795321
524288	0,002717535	0,004169547	0,001324898
1048576	0,005940812	0,004977442	0,004866001
2097152	0,010227134	0,006886052	0,005672662
4194304	0,018487143	0,010423737	0,011656238
8388608	0,034693351	0,022437661	0,017636982
16777216	0,068116241	0,034535603	0,037220747
33554432	0,136646414	0,069476048	0,09779366
67108864	0,26442012	0,1367961	0,13282407

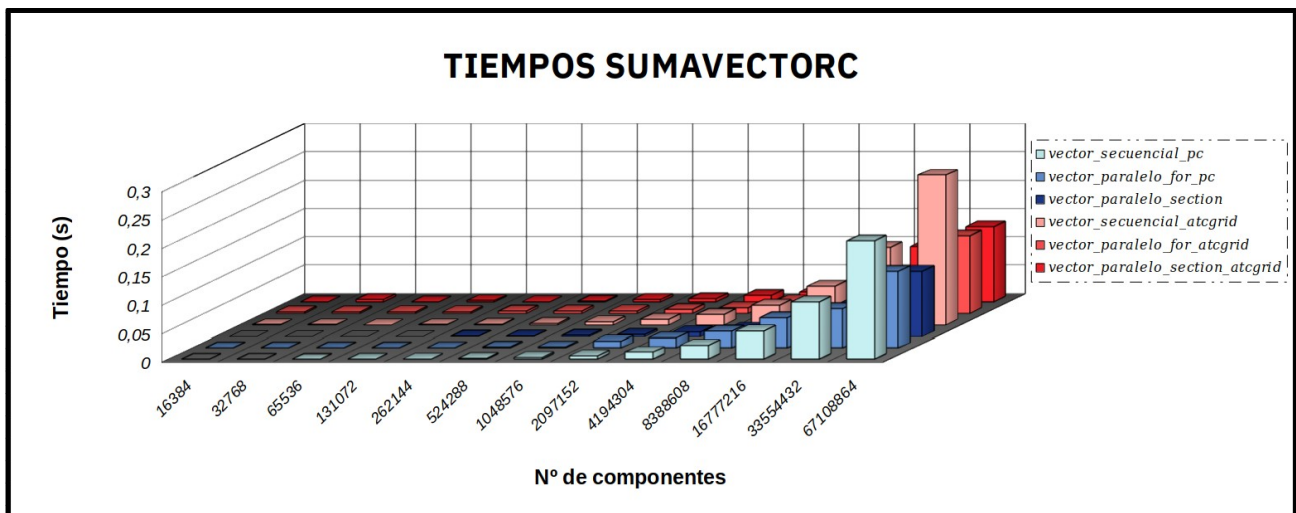


Tabla 2. Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas. Sustituir en el encabezado de la tabla “¿?” por el número de threads utilizados, que debe coincidir con el número de cores físicos del computador que como máximo puede aprovechar el código.

Nº de Componentes	T. secuencial vect. Globales 1 thread/core	T. paralelo (versión for) ¿?threads/cores	T. paralelo (versión sections) ¿?threads/cores
16384			
32768			
65536			
131072			
262144			
524288			
1048576			
2097152			
4194304			
8388608			
16777216			
33554432			
67108864			

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:

Nº de Componentes	Tiempo secuencial vect. Globales 1 thread/core			Tiempo paralelo/version for 24 thread/core		
	<i>Elapsed</i>	<i>CPU-user</i>	<i>CPU-sys</i>	<i>Elapsed</i>	<i>CPU-user</i>	<i>CPU-sys</i>
16384	0m0.005s	0m0.003s	0m0.000s	0m0.014s	0m0.076s	0m0.136s
32768	0m0.003s	0m0.000s	0m0.003s	0m0.012s	0m0.055s	0m0.155s
65536	0m0.003s	0m0.003s	0m0.000s	0m0.012s	0m0.088s	0m0.140s
131072	0m0.004s	0m0.002s	0m0.001s	0m0.012s	0m0.055s	0m0.157s
262144	0m0.006s	0m0.000s	0m0.006s	0m0.013s	0m0.091s	0m0.147s
524288	0m0.009s	0m0.003s	0m0.006s	0m0.015s	0m0.141s	0m0.151s
1048576	0m0.016s	0m0.006s	0m0.010s	0m0.020s	0m0.197s	0m0.178s
2097152	0m0.027s	0m0.012s	0m0.015s	0m0.021s	0m0.204s	0m0.231s
4194304	0m0.047s	0m0.026s	0m0.021s	0m0.027s	0m0.384s	0m0.199s
8388608	0m0.083s	0m0.047s	0m0.036s	0m0.047s	0m0.701s	0m0.252s
16777216	0m0.160s	0m0.088s	0m0.072s	0m0.084s	0m1.241s	0m0.558s
33554432	0m0.316s	0m0.171s	0m0.144s	0m0.161s	0m2.512s	0m1.022s
67108864	0m0.620s	0m0.332s	0m0.285s	0m0.305s	0m4.889s	0m1.864s

El tiempo de CPU, es mayor que el tiempo real en el código paralelo, ya que suma el tiempo de todas las hebras.

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 1 thread/core			Tiempo paralelo/versión for ¿? Threads/cores		
	<i>Elapsed</i>	<i>CPU-user</i>	<i>CPU- sys</i>	<i>Elapsed</i>	<i>CPU-user</i>	<i>CPU- sys</i>
65536						
131072						
262144						
524288						
1048576						
2097152						
4194304						
8388608						
16777216						
33554432						
67108864						