

2º curso / 2º cuatr.
Grado Ing. Inform.
Dobles Grados

Arquitectura de Computadores (AC)

Cuaderno de prácticas.

Bloque Práctico 1. Programación paralela I: Directivas OpenMP

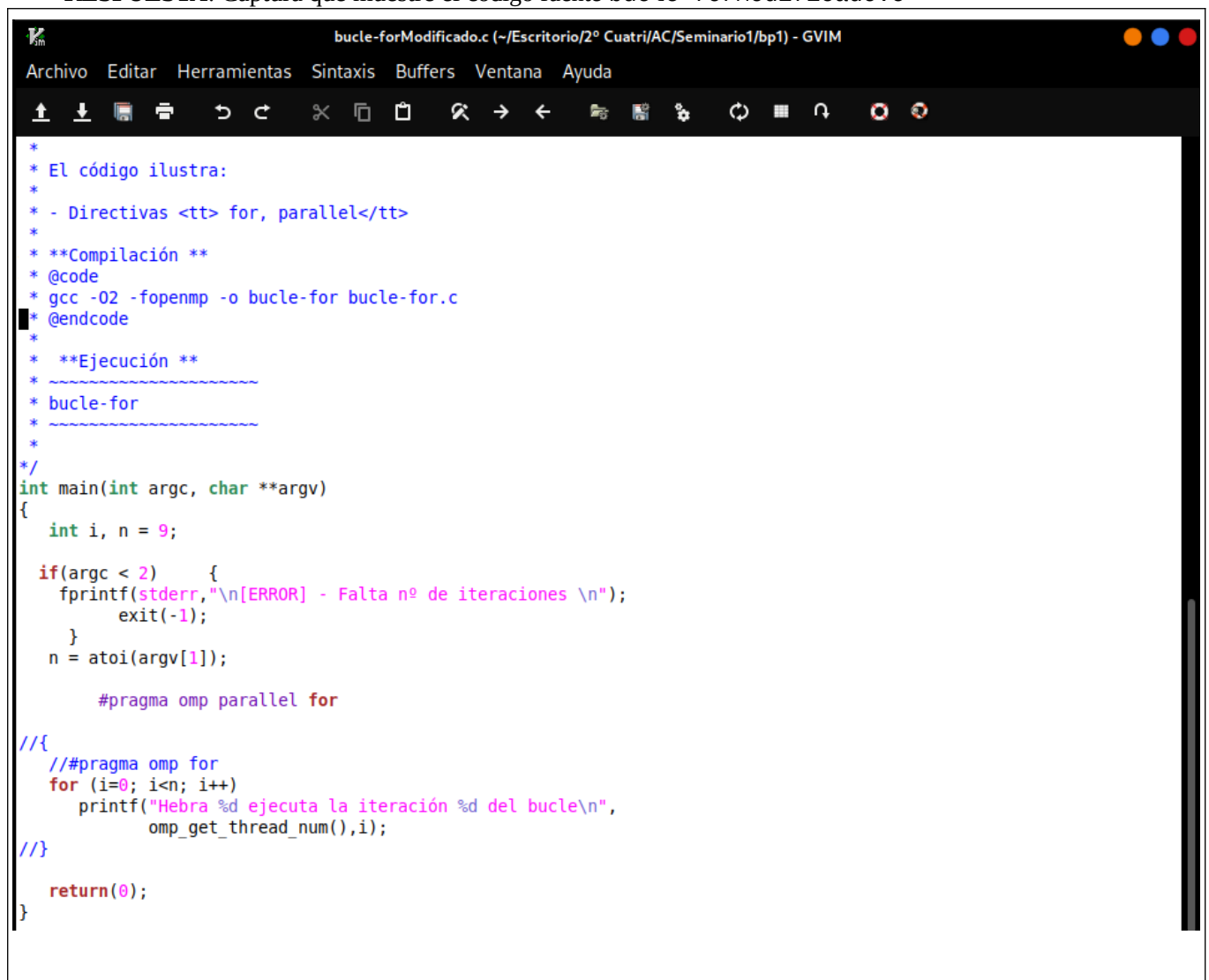
Estudiante (nombre y apellidos):

Grupo de prácticas y profesor de prácticas:

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`



```
*
* El código ilustra:
*
* - Directivas <tt>for, parallel</tt>
*
* **Compilación **
* @code
* gcc -O2 -fopenmp -o bucle-for bucle-for.c
* @endcode
*
* **Ejecución **
* ~~~~~
* bucle-for
* ~~~~~
*
*/
int main(int argc, char **argv)
{
    int i, n = 9;

    if(argc < 2) {
        fprintf(stderr, "\n[ERROR] - Falta nº de iteraciones \n");
        exit(-1);
    }
    n = atoi(argv[1]);

    #pragma omp parallel for

    //{
    // #pragma omp for
    for (i=0; i<n; i++)
        printf("Hebra %d ejecuta la iteración %d del bucle\n",
            omp_get_thread_num(), i);
    //}

    return(0);
}
```

```

icaro@kali: ~/Escritorio/2º Cuatri/AC/Seminario1/bp1
Archivo Acciones Editar Vista Ayuda

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ gcc -O2 -fopenmp bucle-forModificado.c -o bucle-forModificado

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ export OMP_DYNAMIC=FALSE

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ export OMP_NUM_THREADS=8

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ ./bucle-forModificado 8
Hebra 7 ejecuta la iteración 7 del bucle
Hebra 3 ejecuta la iteración 3 del bucle
Hebra 5 ejecuta la iteración 5 del bucle
Hebra 1 ejecuta la iteración 1 del bucle
Hebra 2 ejecuta la iteración 2 del bucle
Hebra 6 ejecuta la iteración 6 del bucle
Hebra 0 ejecuta la iteración 0 del bucle
Hebra 4 ejecuta la iteración 4 del bucle

```

RESPUESTA: Captura que muestre el código fuente sectionsModificado.c

```

sectionsModificado.c (~/Escritorio/2º Cuatri/AC/Seminario1/bp1) - GVIM
Archivo Editar Herramientas Sintaxis Buffers Ventana Ayuda

* El código ilustra:
* - Directivas <tt>sections, parallel</tt>
*
* **Compilación **
* @code
* gcc -O2 -fopenmp -o sections sections.c
* @endcode
*
* **Ejecución **
* ~~~~~
* sections
* ~~~~~
*
*/
void funcA()
{
    printf("En funcA: esta sección la ejecuta la hebra %d\n",
        omp_get_thread_num());
}
void funcB()
{
    printf("En funcB: esta sección la ejecuta la hebra %d\n",
        omp_get_thread_num());
}

int main()
{
    #pragma omp parallel sections
    {
        //#pragma omp sections
        //{
            #pragma omp section
            (void) funcA();

            #pragma omp section
            (void) funcB();
        //}
    }

    return(0);
}

```

```

icaro@kali: ~/Escritorio/2º Cuatri/AC/Seminario1/bp1
Archivo Acciones Editar Vista Ayuda

(icaro@kali)-[~/Escritorio/2º Cuatri/AC/Seminario1/bp1]
$ gcc -O2 -fopenmp sectionsModificado.c -o sectionsModificado

(icaro@kali)-[~/Escritorio/2º Cuatri/AC/Seminario1/bp1]
$ export OMP_DYNAMIC=FALSE

(icaro@kali)-[~/Escritorio/2º Cuatri/AC/Seminario1/bp1]
$ export OMP_NUM_THREADS=2

(icaro@kali)-[~/Escritorio/2º Cuatri/AC/Seminario1/bp1]
$ ./sectionsModificado
En funcA: esta sección la ejecuta la hebra 0
En funcB: esta sección la ejecuta la hebra 1

```

- 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`

```

singleModificado.c (~/Escritorio/2º Cuatri/AC/Seminario1/bp1) - GVIM
Archivo Editar Herramientas Sintaxis Buffers Ventana Ayuda

* gcc -O2 -fopenmp -o single single.c
* @endcode
*
* **Ejecución **
* ~~~~~
* single
* ~~~~~
*/
int main()
{
    int n = 9;
    int i, a, b[n];

    for (i=0; i<n; i++)
        b[i] = -1;
#pragma omp parallel
{
    #pragma omp single
    {
        printf("Introduce valor de inicialización a: "); scanf("%d",&a);
        printf("Single ejecutada por la hebra %d\n",
            omp_get_thread_num());
    }

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

    #pragma omp single
    {
        for(i=0; i < n ; i++)
            printf("b[%d] = %d\t",i,b[i]);
        printf("\n");
        printf("Este segundo single ha sido ejecutado por el thread %d\n", omp_get_thread_num());
    }
}
/*
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);
*/
}

```

```

icaro@kali: ~/Escritorio/2º Cuatri/AC/Seminario1/bp1
Archivo Acciones Editar Vista Ayuda

(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ ls
atomic.c      bucle-forModificado  critical.c      master.c      sectionsModificado  singleModificado.c
barrier.c     bucle-forModificado.c critical_sin.c  parallel.c    sectionsModificado.c

(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ gcc -O2 -fopenmp singleModificado.c -o singleModificado

(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ export OMP_DYNAMIC=FALSE

(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ export OMP_NUM_THREADS=8

(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ ./singleModificado
Introduce valor de inicialización a: 23
Single ejecutada por la hebra 0
b[0] = 23      b[1] = 23      b[2] = 23      b[3] = 23      b[4] = 23      b[5] = 23      b[6] = 23      b[7]
] = 23  b[8] = 23
Este segundo single ha sido ejecutado por el thread 2

```

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`

```

icaro@kali: ~/Escritorio/2º Cuatri/AC/Seminario1/bp1
Archivo Acciones Editar Vista Ayuda

(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ gcc -O2 -fopenmp singleModificado2.c -o singleModificado2

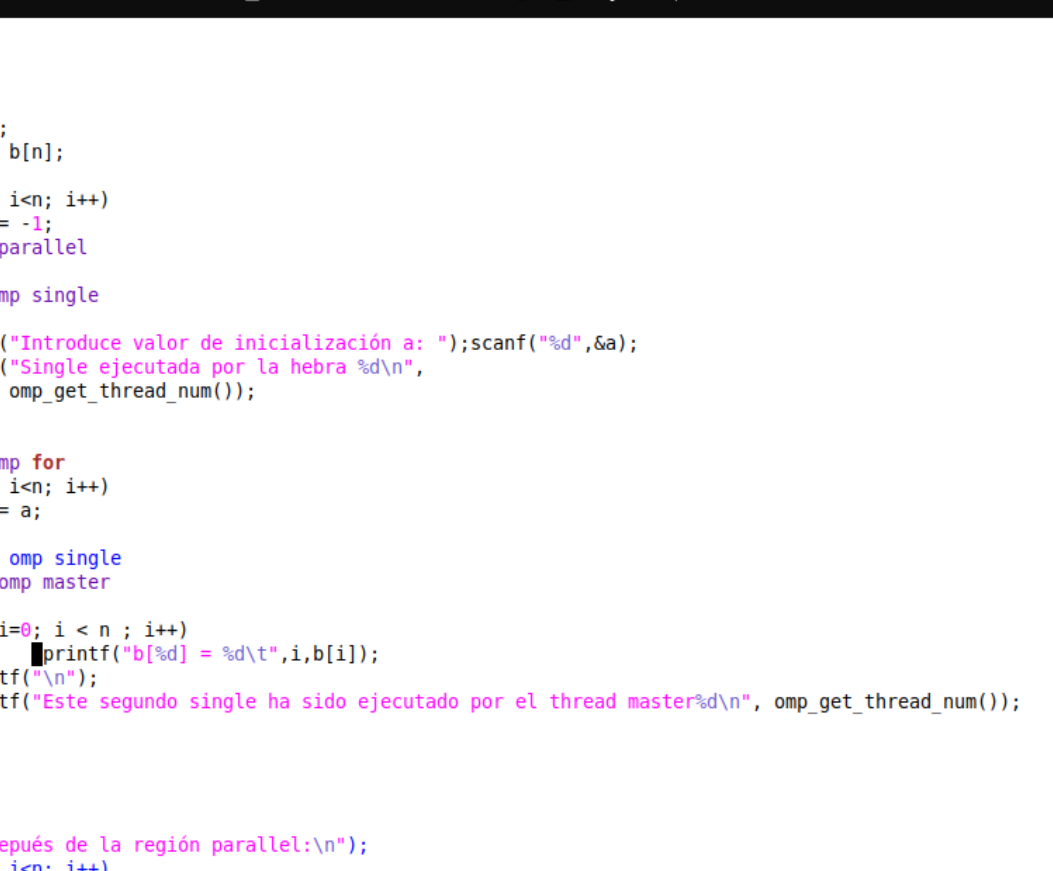
(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ export OMP_DYNAMIC=FALSE

(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ export OMP_NUM_THREADS=8

(icaro@kali)-[~/.../2º Cuatri/AC/Seminario1/bp1]
$ ./singleModificado2
Introduce valor de inicialización a: 23
Single ejecutada por la hebra 0
b[0] = 23      b[1] = 23      b[2] = 23      b[3] = 23      b[4] = 23      b[5] = 23      b[6] = 23      b[7]
= 23  b[8] = 23
Este segundo single ha sido ejecutado por el thread master0

```

CAPTURAS DE PANTALLA:



```
singleModificado2.c (~/Escritorio/2º Cuatri/AC/Seminario1/bp1) - GVIM
Archivo  Editar  Herramientas  Sintaxis  Buffers  Ventana  Ayuda

*
*/
int main()
{
    int n = 9;
    int i, a, b[n];

    for (i=0; i<n; i++)
        b[i] = -1;
#pragma omp parallel
{
    #pragma omp single
    {
        printf("Introduce valor de inicialización a: ");scanf("%d",&a);
        printf("Single ejecutada por la hebra %d\n",
            omp_get_thread_num());
    }

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

    /*#pragma omp single
    #pragma omp master
    {
        for(i=0; i < n ; i++)
            printf("b[%d] = %d\t",i,b[i]);
        printf("\n");
        printf("Este segundo single ha sido ejecutado por el thread master%d\n", omp_get_thread_num());
    }
}
*/
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);
*/
}
```

RESPUESTA A LA PREGUNTA:

La diferencia con respecto a la ejecución del ejercicio 2 es que el thread que imprime los resultados de la ejecución del programa dentro de la construcción parallel es el thread master, es decir la hebra numero 0

Esto no influye sobre los resultados del programa

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

Viene en las diapositivas un ejemplo de `master.c` con la barrera eliminada

```

icaro@kali: ~/Escritorio/2º Cuatri/AC/Seminario1/bp1
Archivo Acciones Editar Vista Ayuda

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ gcc -O2 -fopenmp masterconbarrera.c -o masterconbarrera

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ export OMP_DYNAMIC=FALSE

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ export OMP_NUM_THREADS=3

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ ./masterconbarrera 6
Hebra 0 suma de a[0]=0 sumalocal=0
Hebra 0 suma de a[1]=1 sumalocal=1
Hebra 2 suma de a[4]=4 sumalocal=4
Hebra 2 suma de a[5]=5 sumalocal=9
Hebra 1 suma de a[2]=2 sumalocal=2
Hebra 1 suma de a[3]=3 sumalocal=5
Thread master=0 imprime suma=15

```

```

icaro@kali: ~/Escritorio/2º Cuatri/AC/Seminario1/bp1
Archivo Acciones Editar Vista Ayuda

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ gcc -O2 -fopenmp mastersinbarrera.c -o mastersinbarrera

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ export OMP_DYNAMIC=FALSE

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ export OMP_NUM_THREADS=3

(icaro@kali)-[~/2º Cuatri/AC/Seminario1/bp1]
$ ./mastersinbarrera 6
Hebra 0 suma de a[0]=0 sumalocal=0
Hebra 0 suma de a[1]=1 sumalocal=1
Hebra 2 suma de a[4]=4 sumalocal=4
Hebra 2 suma de a[5]=5 sumalocal=9
Hebra 1 suma de a[2]=2 sumalocal=2
Hebra 1 suma de a[3]=3 sumalocal=5
Thread master=0 imprime suma=1

```

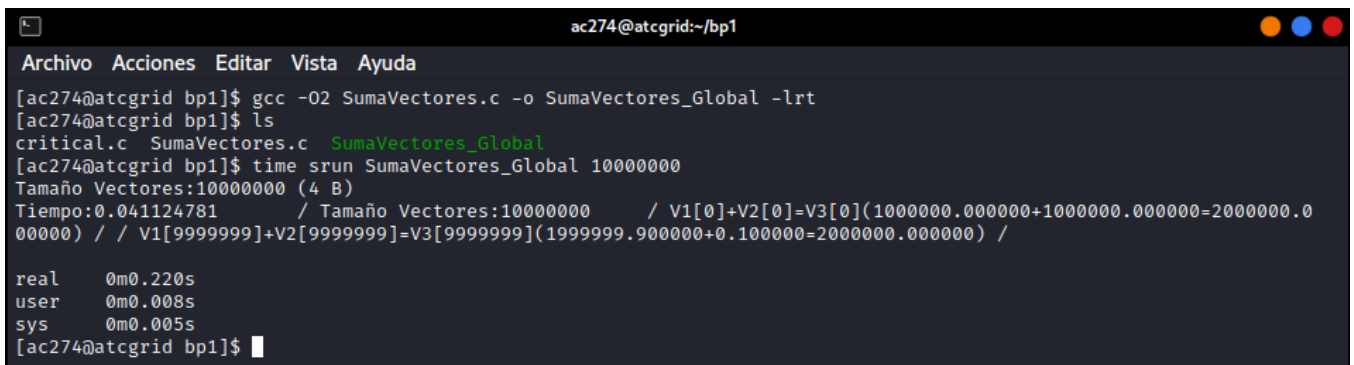
Se suman las componentes paralelizando las threads las sumas. La suma total es la suma de las sumas parciales y se actualiza la variable global compartida, para ello se tiene que asegurar que se accede a ella de forma ordenada con crítica, en el ejemplo se usa `atomic` que es lo mismo. Como con `parallel for` hay barrera implícita, pues entonces cuando se calcule la suma se hará correctamente.

Pero en el ejemplo de `master.c` se quiere hacer dentro del código que paraleliza, entonces como `atomic` no tiene barrera implícita, entonces tenemos que usar `barrier` para saber que se respeta la zona crítica y asegurarnos de que suma tiene sumado todos los valores de las sumas locales. De no poner `barrier`, la zona crítica se va a respetar pero se puede mostrar por pantalla el valor de la suma total faltando por sumar algunas sumas parciales, es decir, no siempre la salida va a ser correcta, por eso usamos el `barrier`, para asegurarnos de que se hacen la suma.

Resto de ejercicios (usar en atcgrid la cola ac a no ser que se tenga que usar atcgrid4)

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:



```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
[ac274@atcgrid bp1]$ gcc -O2 SumaVectores.c -o SumaVectores_Global -lrt
[ac274@atcgrid bp1]$ ls
critical.c SumaVectores.c SumaVectores_Global
[ac274@atcgrid bp1]$ time srun SumaVectores_Global 10000000
Tamaño Vectores:10000000 (4 B)
Tiempo:0.041124781 / 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    0m0.220s
user    0m0.008s
sys     0m0.005s
[ac274@atcgrid bp1]$
```

RESPUESTA:

`time srun SumaVectoresC_global 10000000`

La suma de los tiempos de CPU del usuario y del sistema es menor que el tiempo real porque el tiempo que falta es el asociado a las esperas debidas a E/S o asociadas a la ejecución de otros programas

real = 0m0.220s (aquí se añaden además las esperas del sistema)

user = 0m0.008s y sys = 0m0.005s

6. Generar el código ensamblador a partir del programa secuencial C del Listado 1 (ver cuaderno de BP0) 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. Incorporar **el código ensamblador de la parte de la suma de vectores** (no de todo el programa) 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):

RESPUESTA: cálculo de los MIPS y los MFLOPS

$MIPS = NI / (T_{cpu} + 10^6)$

$MFLOPS = n^{FP} / T_{cpu} + 10^6$

Código ensamblador -> `gcc -O2 -S SumaVectores.c -lrt` (la S en mayúscula)

Todo lo que hay entre 2 llamadas a `clock_gettime` es la suma además que aparece la etiqueta del bucle (L6)

Consultando un manual del código ensamblador observamos que las operaciones en coma flotante son `movsd` y `addsd`

```
movsd v1(%rax,8), %xmm0 // (3)
addsd v2(%rax,8), %xmm0
movsd %xmm0, v3(%rax,8)
```

```
N(10 o 10000000)
Antes -> hay 1 // 3 instrucciones
Luego bucle hay 2
bucle -> 6 instrucciones * N
movsd v1(%rax,8), %xmm0
addsd v2(%rax,8), %xmm0
movsd %xmm0, v3(%rax,8)
addq $1, %rax
cmpl %eax, %ebp
ja .L6
```

hay 3 operaciones en coma flotante `MOVSD ADDSD MOVSD`

$NI = 3 + 6 * N$
 $n^{FP} = 3 * N$

el bucle se comprende desde salto `.L6` hasta el principio de este -> `movsd`. Por lo tanto tendremos que multiplicar por 6 el número de componentes. Le sumamos 3 a la expresión ya que hay 3 instrucciones fuera del bucle y entre `clock_gettime`

Para 10 componentes

$NI = 6 * 10 + 3 = 63$
 $n^{FP} = 3 * 10$
 Tiempo (T_{cpu}): 0.000404725

$MIPS = 63 / (0.000404725 * 10^6) = 0.155661$ MIPS (porque dividido por 10^6 , si fuese por 10^9 sería GMIPS/GFLOPS)

$MFLOPS = 30 / (0.000404725 * 10^6) = 0.0741244$ MFLOPS

Para 10000000

$$NI = 6 * 10000000 + 3 = 60000003$$

$$nFP = 3 * 10000000 = 30000000$$

$$\text{Tiempo} = 0.041432411$$

$$\text{MIPS} = 60000003 / (0.041432411 * 10^6) = 1448.141722 \text{ MIPS}$$

$$\text{MFLOPS} = 30000000 / (0.041432411 * 10^6) = 724.11445 \text{ MFLOPS}$$

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

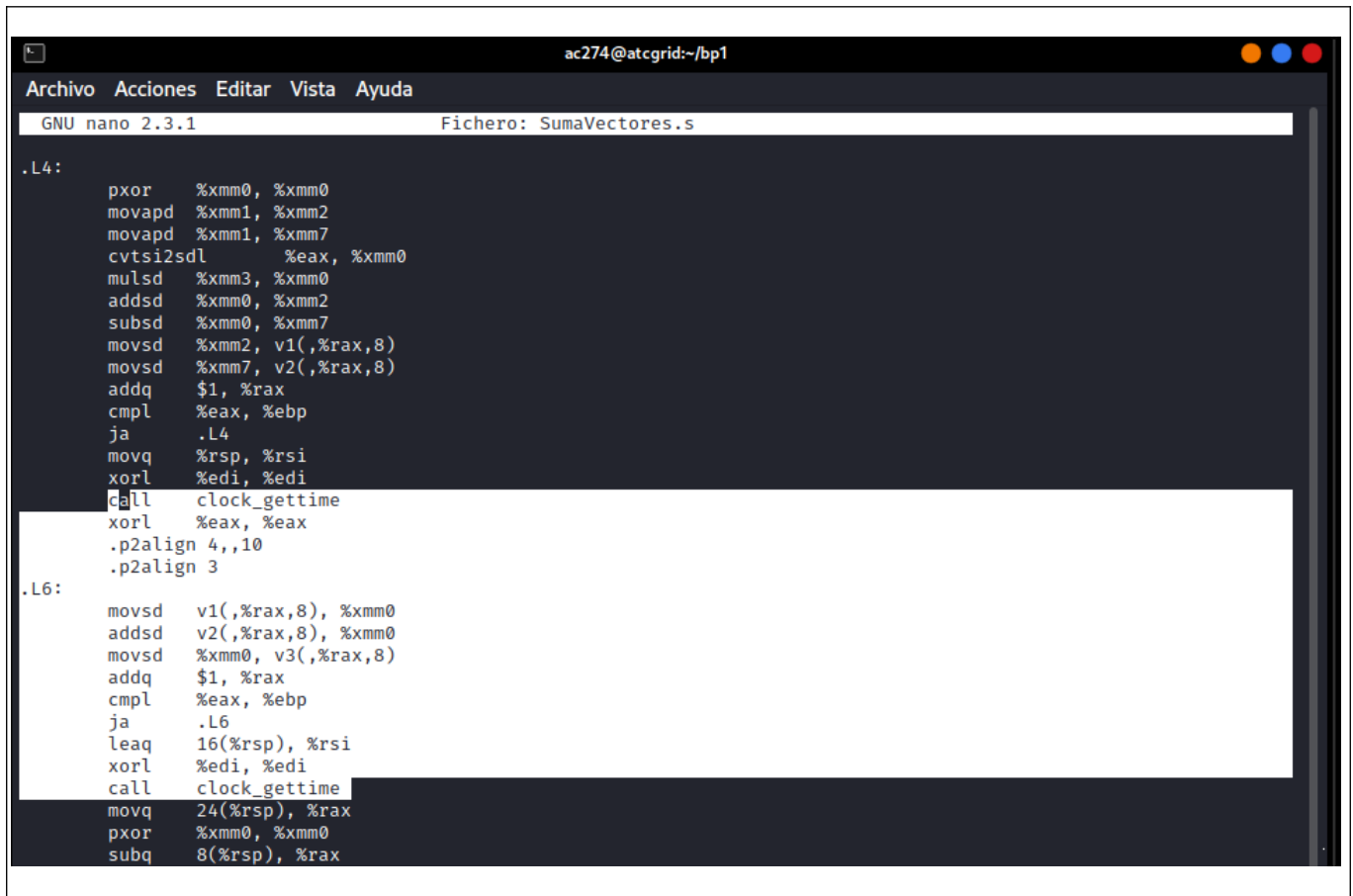
```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
[ac274@atcgrid bp1]$ gcc -O2 SumaVectores.c -o SumaVectores_Global -lrt
[ac274@atcgrid bp1]$ ls
critical.c SumaVectores.c SumaVectores_Global
[ac274@atcgrid bp1]$ time srun SumaVectores_Global 10000000
Tamaño Vectores:10000000 (4 B)
Tiempo:0.041124781 / 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    0m0.220s
user    0m0.008s
sys     0m0.005s
[ac274@atcgrid bp1]$
```

```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
[ac274@atcgrid bp1]$ gcc -O2 -S SumaVectores.c -lrt
[ac274@atcgrid bp1]$ ls
critical.c SumaVectores.c SumaVectores_Global SumaVectores.s
[ac274@atcgrid bp1]$ time srun SumaVectores_Global 10000000
Tamaño Vectores:10000000 (4 B)
Tiempo:0.041432411 / 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    0m0.237s
user    0m0.006s
sys     0m0.005s
[ac274@atcgrid bp1]$ time srun SumaVectores_Global 10
Tamaño Vectores:10 (4 B)
Tiempo:0.000404725 / 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) /

real    0m0.110s
user    0m0.010s
sys     0m0.003s
[ac274@atcgrid bp1]$
```



```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
GNU nano 2.3.1 Fichero: SumaVectores.s

.L4:
    pxor    %xmm0, %xmm0
    movapd  %xmm1, %xmm2
    movapd  %xmm1, %xmm7
    cvtsi2sd %eax, %xmm0
    mulsd   %xmm3, %xmm0
    addsd   %xmm0, %xmm2
    subdsd  %xmm0, %xmm7
    movsd   %xmm2, v1(%rax,8)
    movsd   %xmm7, v2(%rax,8)
    addq    $1, %rax
    cmpl    %eax, %ebp
    ja      .L4
    movq    %rsp, %rsi
    xorl    %edi, %edi
    call    clock_gettime
    xorl    %eax, %eax
    .p2align 4,,10
    .p2align 3

.L6:
    movsd   v1(%rax,8), %xmm0
    addsd   v2(%rax,8), %xmm0
    movsd   %xmm0, v3(%rax,8)
    addq    $1, %rax
    cmpl    %eax, %ebp
    ja      .L6
    leaq    16(%rsp), %rsi
    xorl    %edi, %edi
    call    clock_gettime
    movq    24(%rsp), %rax
    pxor    %xmm0, %xmm0
    subq    8(%rsp), %rax
```

Lo seleccionado corresponde al bucle

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 $N = 11$); (4) se debe imprimir el tamaño de los vectores y el número de hilos; (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 `sp-OpenMP-for.c`

```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda

#ifdef VECTOR_DYNAMIC
double *v1, *v2, *v3;

// variables start y end
//double start,end;

v1 = (double*) malloc(N*sizeof(double)); // malloc necesita el tamaño en bytes
v2 = (double*) malloc(N*sizeof(double));
v3 = (double*) malloc(N*sizeof(double));
if ((v1 == NULL) || (v2 == NULL) || (v3 == NULL)) {
    printf("No hay suficiente espacio para los vectores \n");
    exit(-2);
}
#endif

//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; //Se puede usar drand48() para generar los valores de forma aleatoria
    (drand48_r() para una versión paralela)
}

// clock_gettime(CLOCK_REALTIME,&cgt1);
double start = omp_get_wtime();

//Calcular suma de vectores
#pragma omp parallel for
for(i=0; i<N; i++){
    //para comprobar que se hace la parametrización
    //printf("iteracion: %d hebra: %d ",i, omp_get_thread_num());

    v3[i] = v1[i] + v2[i];
}

//clock_gettime(CLOCK_REALTIME,&cgt2);
double end = omp_get_wtime();

//Calculo del tiempo
//ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
// (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
ncgt = end - start;

//Imprimir resultado de la suma y el tiempo de ejecución

// imprimo el hilo que lo ha ejecutado
//printf ("Se ha ejecutado el thread %d\n", omp_get_max_num());

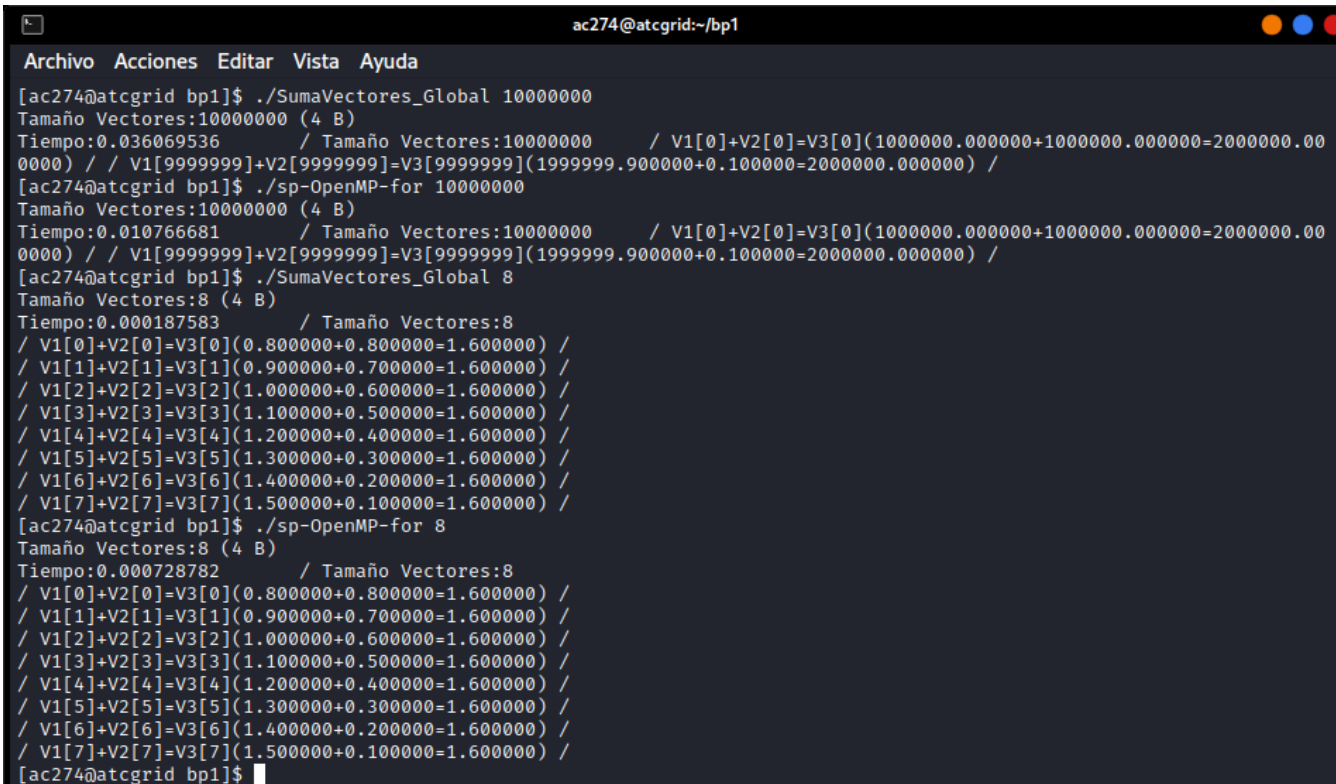
if (N<10) {
    printf("Tiempo:%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
    for(i=0; i<N; i++)
        printf("/ V1[%d]+V2[%d]=V3[%d](%8.6f+%8.6f=%8.6f) /\n",
            i,i,v1[i],v2[i],v3[i]);
}
else
    printf("Tiempo:%11.9f\t / Tamaño Vectores:%u\t/ V1[0]+V2[0]=V3[0](%8.6f+%8.6f=%8.6f) / / V1[%d]+V2[%d]=V3[%d](%8.6f+%8.6f=%8.6f) /\n",
        ncgt,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);

"sp-OpenMP-for.c" 135L, 4447C 103,2 90%
```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

```
[ac274@atcgrid bp1]$ ./sp-OpenMP-for 10
Tamaño Vectores:10 (4 B)
iteracion: 2 hebra: 1 iteracion: 0 hebra: 0 iteracion: 9 hebra: 7 iteracion: 8 hebra: 6 iteracion: 5 hebra: 3 iteracion:
7 hebra: 5 iteracion: 6 hebra: 4 iteracion: 4 hebra: 2 iteracion: 3 hebra: 1 iteracion: 1 hebra: 0 Tiempo:0.000716590
/ 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=2.00
0000) /
```



```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
[ac274@atcgrid bp1]$ ./SumaVectores_Global 10000000
Tamaño Vectores:10000000 (4 B)
Tiempo:0.036069536 / Tamaño Vectores:10000000 / V1[0]+V2[0]=V3[0](1000000.000000+1000000.000000=2000000.00
0000) / / V1[9999999]+V2[9999999]=V3[9999999](1999999.900000+0.100000=2000000.000000) /
[ac274@atcgrid bp1]$ ./sp-OpenMP-for 10000000
Tamaño Vectores:10000000 (4 B)
Tiempo:0.010766681 / Tamaño Vectores:10000000 / V1[0]+V2[0]=V3[0](1000000.000000+1000000.000000=2000000.00
0000) / / V1[9999999]+V2[9999999]=V3[9999999](1999999.900000+0.100000=2000000.000000) /
[ac274@atcgrid bp1]$ ./SumaVectores_Global 8
Tamaño Vectores:8 (4 B)
Tiempo:0.000187583 / 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) /
[ac274@atcgrid bp1]$ ./sp-OpenMP-for 8
Tamaño Vectores:8 (4 B)
Tiempo:0.000728782 / 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) /
[ac274@atcgrid bp1]$
```

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, $N = 8$); (4) se debe imprimir el tamaño de los vectores y el número de hilos; (5) 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 `sp-OpenMP-sections.c`

```

ac274@atcgriid:~/bp1
Archivo Acciones Editar Vista Ayuda
// v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1; //Se puede usar drand48() para generar los valores de forma aleatori
a (drand48_r() para una versión paralela)
//}

#pragma omp parallel sections private(i)
{
    #pragma omp section
    for (i=0; i < N/3 ; i++){
        v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
    }

    #pragma omp section
    for (i=N/3; i < 2*N/3; i++){
        v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
    }

    #pragma omp section
    for (i=2*N/3; i < N; i++){
        v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
    }
}

//clock_gettime(CLOCK_REALTIME,&cgt1);
double start = omp_get_wtime();

//Calcular suma de vectores
//for(i=0; i<N; i++){
//    printf("Suma | iteración:%d | hebra:%d\n",i,omp_get_thread_num());
//    v3[i] = v1[i] + v2[i];
//}

#pragma omp parallel sections private(i)
{
    #pragma omp section
    for (i=0; i < N/3 ; i++){
        printf("Suma | iteración:%d | hebra:%d\n",i,omp_get_thread_num());
        v3[i] = v1[i] + v2[i];
    }

    #pragma omp section
    for (i=N/3; i < 2*N/3; i++){
        printf("Suma | iteración:%d | hebra:%d\n",i,omp_get_thread_num());
        v3[i] = v1[i] + v2[i];
    }

    #pragma omp section
    for (i=2*N/3; i < N; i++){
        printf("Suma | iteración:%d | hebra:%d\n",i,omp_get_thread_num());
        v3[i] = v1[i] + v2[i];
    }
}

// Ahora uso omp_get_wtime()
//clock_gettime(CLOCK_REALTIME,&cgt2);
//ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
//    (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
double end = omp_get_wtime();
ncgt= end -start;

```

143,1 80%

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

```

[ac274@atcgrid bp1]$ ls
critical.c      sp-OpenMP-for.c      sp-OpenMP-sections.c  SumaVectores_Global
sp-OpenMP-for  sp-OpenMP-sections  SumaVectores.c        SumaVectores.s
[ac274@atcgrid bp1]$ ./sp-OpenMP-sections 8
Tamaño Vectores:8 (4 B)
Suma | iteración:5 | hebra:1
Suma | iteración:2 | hebra:2
Suma | iteración:0 | hebra:5
Suma | iteración:6 | hebra:1
Suma | iteración:7 | hebra:1
Suma | iteración:3 | hebra:2
Suma | iteración:4 | hebra:2
Suma | iteración:1 | hebra:5
Tiempo:0.000211415 / 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) /
[ac274@atcgrid bp1]$ ./sp-OpenMP-sections 11
Tamaño Vectores:11 (4 B)
Suma | iteración:3 | hebra:1
Suma | iteración:7 | hebra:5
Suma | iteración:0 | hebra:4
Suma | iteración:8 | hebra:5
Suma | iteración:9 | hebra:5
Suma | iteración:10 | hebra:5
Suma | iteración:4 | hebra:1
Suma | iteración:5 | hebra:1
Suma | iteración:6 | hebra:1
Suma | iteración:1 | hebra:4
Suma | iteración:2 | hebra:4
Tiempo:0.000195169 / Tamaño Vectores:11 / V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000) / / V1[10]+V2[10]=V
3[10](2.100000+0.100000=2.200000) /
[ac274@atcgrid bp1]$
  
```

gcc 02 fopenmp spOpenMP-sections-c -o spOpenMP-sections -lrt

Lo que he hecho es dividir el vector en 3 partes iguales, uno que va de 0 a $n/3$, otro de $n/3$ a $2*n/3$ y uno último que va de $2*n/3$ a n , cada uno en su correspondiente section, englobado de un parallel y un sections private(i), private(i) para que cada hebra i sea privada.

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. NOTA: Al contestar piense sólo en el código, no piense en el computador en el que lo va a ejecutar.

RESPUESTA:

En el ejercicio 7, usando las directivas for y parallel, el reparto de trabajo entre threads va según las iteraciones del bucle, es decir se reparte de forma dinámica, depende del valor de OMP_NUM_THREADS, podríamos usar cuantos quisiésemos siendo este inferior al numero de cores de la maquina, en mi caso en mi portátil, el máximo son 8 hebras. Por otro lado, con parallel y section, aquí la asignación de hebras no es dinámica, como mucho se van a usar 3 en mi caso, por que uso 3 section, es decir 3 bloques.

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. Escribir un script para realizar las ejecuciones necesarias utilizando como base el script del seminario de BP0 (se deben imprimir en el script al menos las variables de entorno que ya se imprimen en el script de BP0). 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. Observar que el número de componentes en la tabla llega hasta **67108864**.

RESPUESTA: Captura del script implementado sp-OpenMP-script10.sh

```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
#!/bin/bash

echo "Id. usuario del trabajo: $SLURM_JOB_USER"
echo "Id. del trabajo: $SLURM_JOBID"
echo "Nombre del trabajo especificado por usuario: $SLURM_JOB_NAME"
echo "Directorio de trabajo (en el que se ejecuta el script): $SLURM_SUBMIT_DIR"
echo "Cola: $SLURM_JOB_PARTITION"
echo "Nodo que ejecuta este trabajo: $SLURM_SUBMIT_HOST"
echo "Nº de nodos asignados al trabajo: $SLURM_JOB_NUM_NODES"
echo "Nodos asignados al trabajo: $SLURM_JOB_NODELIST"
echo "CPUs por nodo: $SLURM_JOB_CPUS_PER_NODE"

# instrucciones del script para ejecutar el código:
echo -e Ejecucion de SumaVectores_Global, con for paralelizado y sections

for ((i=16384;i≤67108864;i=i*2))
do
    echo -e "\nEjecutando con for con $i componentes"
    ./sp-OpenMP-for $i

    echo -e "\nEjecutando con sections con $i componentes"
    ./sp-OpenMP-sections $i

    echo -e "\nEjecutando secuencial, sin paralelismo, con $i componentes"
    ./SumaVectores_Global $i
done
```

(RECUERDE ADJUNTAR
LOS CÓDIGOS AL .ZIP)

CAPTURAS DE PANTALLA (mostrar la ejecución
en atcgrid – envío(s) a la
cola):

```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
[ac274@atcgrid bp1]$ sbatch -pac -n1 -c12 --hint=nomultithread sp-OpenMP-script10.sh
Submitted batch job 137797
[ac274@atcgrid bp1]$ cat slurm-137797.out
Id. usuario del trabajo: ac274
Id. del trabajo: 137797
Nombre del trabajo especificado por usuario: sp-OpenMP-script10.sh
Directorio de trabajo (en el que se ejecuta el script): /home/ac274/bp1
Cola: ac
Nodo que ejecuta este trabajo:
Nº de nodos asignados al trabajo: 1
Nodos asignados al trabajo: atcgrid1
CPUs por nodo:
Ejecucion de SumaVectores_Global, con for paralelizado y sections

Ejecutando con for con 16384 componentes
Tamaño Vectores:16384 (4 B)
Tiempo:0.004894710 / Tamaño Vectores:16384 / V1[0]+V2[0]-V3[0](1638.400000+1638.400000-3276.800000) /
/ V1[16383]+V2[16383]-V3[16383](3276.700000+0.100000-3276.800000) /

Ejecutando con sections con 16384 componentes
Tamaño Vectores:16384 (4 B)
Tiempo:0.002430262 / Tamaño Vectores:16384 / V1[0]+V2[0]-V3[0](1638.400000+1638.400000-3276.800000) /
/ V1[16383]+V2[16383]-V3[16383](3276.700000+0.100000-3276.800000) /

Ejecutando secuencial, sin paralelismo, con 16384 componentes
Tamaño Vectores:16384 (4 B)
Tiempo:0.000250190 / Tamaño Vectores:16384 / V1[0]+V2[0]-V3[0](1638.400000+1638.400000-3276.800000) /
/ V1[16383]+V2[16383]-V3[16383](3276.700000+0.100000-3276.800000) /

Ejecutando con for con 32768 componentes
Tamaño Vectores:32768 (4 B)
Tiempo:0.003049414 / Tamaño Vectores:32768 / V1[0]+V2[0]-V3[0](3276.800000+3276.800000-6553.600000) /
/ V1[32767]+V2[32767]-V3[32767](6553.500000+0.100000-6553.600000) /
```

sbatch -pac -n1 -c12 -hint-nomultithread sp-OpenMP-script10.sh**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 y cores lógicos utilizados.

ATCGRID			
Nº de Componentes	T. secuencial vect. Globales 1 thread=core	T. paralelo (versión for) 12 threads = cores lógicos = cores físicos	T. paralelo (versión sections) 3 threads = cores lógicos = cores físicos
16384	0.000250190	0.004894710	0.002430262
32768	0.000287713	0.003049414	0.000445560
65536	0.000409360	0.003112856	0.000614467
131072	0.000586898	0.003311381	0.000485545
262144	0.001363127	0.003709209	0.000988526
524288	0.002663825	0.004438639	0.003821275
1048576	0.005363612	0.006938809	0.004535850
2097152	0.009397462	0.007702139	0.007329708
4194304	0.017581251	0.011256242	0.015493248
8388608	0.034329310	0.016561990	0.029728538
16777216	0.065527076	0.029912879	0.050662792
33554432	0.127142432	0.058721667	0.070137475
67108864	0.127624406	0.058373666	0.082395221

MI PC (8 cores lógicos)			
Nº de Componentes	T. secuencial vect. Globales 1 thread=core	T. paralelo (versión for) 8 threads = cores lógicos = cores físicos	T. paralelo (versión sections) 3 threads = cores lógicos = cores físicos
16384	0.000355235	0.003239780	0.001818012
32768	0.000413095	0.002301383	0.001506381
65536	0.000785172	0.002589026	0.001013683
131072	0.001104517	0.003354210	0.000872541
262144	0.001683162	0.005560199	0.001518545
524288	0.002962797	0.005155179	0.002881710
1048576	0.005481147	0.006336064	0.004737374
2097152	0.010590638	0.008911975	0.007142738
4194304	0.017966775	0.016914798	0.017502581
8388608	0.035194406	0.031320925	0.041720546
16777216	0.075202113	0.063476719	0.069497655
33554432	0.091203081	0.122715876	0.071366936
67108864	0.091992160	0.080146666	0.073767370

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 (que debe coincidir con el número de cores físicos y lógicos) que usan los códigos. Escribir un script para realizar las ejecuciones necesarias utilizando como base el script del seminario de BP0 (se deben imprimir en el script al menos las variables de entorno que ya se imprimen en el script de BP0) ¿El tiempo de CPU que se obtiene es mayor o igual que el tiempo real (*elapsed*)? Justifique la respuesta.

RESPUESTA: Captura del script implementado `sp-OpenMP-script11.sh`

```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
#!/bin/bash

echo "Id. usuario del trabajo: $SLURM_JOB_USER"
echo "Id. del trabajo: $SLURM_JOBID"
echo "Nombre del trabajo especificado por usuario: $SLURM_JOB_NAME"
echo "Directorio de trabajo (en el que se ejecuta el script): $SLURM_SUBMIT_DIR"
echo "Cola: $SLURM_JOB_PARTITION"
echo "Nodo que ejecuta este trabajo: $SLURM_SUBMIT_HOST"
echo "Nº de nodos asignados al trabajo: $SLURM_JOB_NUM_NODES"
echo "Nodos asignados al trabajo: $SLURM_JOB_NODELIST"
echo "CPUs por nodo: $SLURM_JOB_CPUS_PER_NODE"

# instrucciones del script para ejecutar el código:
echo -e Ejecución de SumaVectores_Global, con for paralelizado y sections

for ((i=8288608;i<=67108864;i=i*2))
do
    echo -e "\nEjecutando con for con $i componentes"
    time ./sp-OpenMP-for $i

    echo -e "\nEjecutando secuencial, sin paralelismo, con $i componentes"
    time ./SumaVectores_Global $i
done
```

```
ac274@atcgrid:~/bp1
Archivo Acciones Editar Vista Ayuda
[ac274@atcgrid bp1]$ sbatch -pac -n1 -c12 --hint=nomultithread sp-OpenMP-script11.sh
Submitted batch job 137852
[ac274@atcgrid bp1]$ cat slurm-137852.out
Id. usuario del trabajo: ac274
Id. del trabajo: 137852
Nombre del trabajo especificado por usuario: sp-OpenMP-script11.sh
Directorio de trabajo (en el que se ejecuta el script): /home/ac274/bp1
Cola: ac
Nodo que ejecuta este trabajo:
Nº de nodos asignados al trabajo: 1
Nodos asignados al trabajo: atcgrid1
CPUs por nodo:
Ejecución de SumaVectores_Global, con for paralelizado y sections

Ejecutando con for con 8288608 componentes
Tamaño Vectores:8288608 (4 B)
Tiempo:0.015966739 / Tamaño Vectores:8288608 / V1[0]+V2[0]=V3[0](828860.800000+828860.800000=1657721.600000) / / V1[8288607]+V2[8288607]=V3[8288607](1657721.500000+0.100000=1657721.600000) /

real    0m0.135s
user    0m0.559s
sys     0m0.300s

Ejecutando secuencial, sin paralelismo, con 8288608 componentes
Tamaño Vectores:8288608 (4 B)
Tiempo:0.033917765 / Tamaño Vectores:8288608 / V1[0]+V2[0]=V3[0](828860.800000+828860.800000=1657721.600000) / / V1[8288607]+V2[8288607]=V3[8288607](1657721.500000+0.100000=1657721.600000) /

real    0m0.144s
user    0m0.051s
sys     0m0.035s

Ejecutando con for con 16577216 componentes
Tamaño Vectores:16577216 (4 B)
Tiempo:0.029416483 / Tamaño Vectores:16577216 / V1[0]+V2[0]=V3[0](1657721.600000+1657721.600000=3315443.200000) / / V1[16577215]+V2[16577215]=V3[16577215](3315443.100000+0.100000=3315443.200000) /

real    0m0.092s
user    0m1.104s
sys     0m0.473s
```

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 Componentes	Tiempo secuencial vect. Globales 1 thread = 1 core lógico = 1 core físico			Tiempo paralelo/versión for 12 Threads = cores lógicos=cores físicos		
	<i>Elapsed</i>	<i>CPU-user</i>	<i>CPU- sys</i>	<i>Elapsed</i>	<i>CPU-user</i>	<i>CPU- sys</i>
8388608	real	0m0.144s		real	0m0.135s	
	user	0m0.051s		user	0m0.559s	
	sys	0m0.035s		sys	0m0.300s	
16777216	real	0m0.177s		real	0m0.092s	
	user	0m0.075s		user	0m1.104s	
	sys	0m0.080s		sys	0m0.473s	
33554432	real	0m0.335s		real	0m0.157s	
	user	0m0.154s		user	0m2.103s	
	sys	0m0.150s		sys	0m0.861s	
67108864	real	0m0.327s		real	0m0.159s	
	user	0m0.148s		user	0m2.148s	
	sys	0m0.156s		sys	0m0.972s	