

2º curso / 2º cuatr.

Grados
Ingeniería
Informática

Arquitectura de Computadores (AC)

1.1.1 Cuaderno de prácticas.

1.1.2 Bloque Práctico 1. Programación paralela I: Directivas OpenMP

Estudiante (nombre y apellidos): Antonio Javier Rodríguez Romero

Grupo de prácticas y profesor de prácticas: Álvaro Martínez. DGIM1

2 Parte I. 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`

```
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
    {
        #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);
}
```

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

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

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

    return(0);
}
```

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`

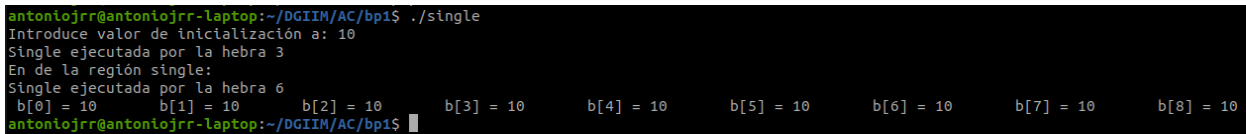
```
int main()
{
    int n = 9;
    int i, a, b[n];

    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
    {
        printf("En la región single:\n");
        printf("Single ejecutada 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");
    }
}
return(0);
}
```

CAPTURAS DE PANTALLA:



antoniojrr@antoniojrr-laptop:~/DGIIM/AC/bp1\$./single
Introduce valor de inicialización a: 10
Single ejecutada por la hebra 3
En de la región single:
Single ejecutada por la hebra 6
b[0] = 10 b[1] = 10 b[2] = 10 b[3] = 10 b[4] = 10 b[5] = 10 b[6] = 10 b[7] = 10 b[8] = 10
antoniojrr@antoniojrr-laptop:~/DGIIM/AC/bp1\$

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`

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

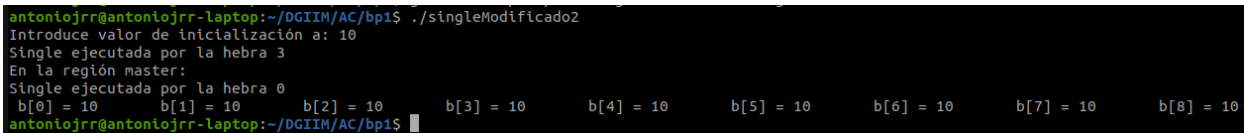
    for (i=0; i<n; i++)
        b[i] = -1;
#pragma omp parallel
{
    #pragma omp single
    {
        printf("Introduce valor de inicialización a: ");scanf("%d",&a);
        printf("Single ejecutada por la hebra %d\n",
            omp_get_thread_num());
    }
}
```

```

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

#pragma omp master
{
    printf("En la región master:\n");
    printf("Single ejecutada 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");
}
return(0);
}

```

CAPTURAS DE PANTALLA:


```

antoniojrr@antoniojrr-laptop:~/DGIIM/AC/bp1$ ./singleModificado2
Introduce valor de inicialización a: 10
Single ejecutada por la hebra 3
En la región master:
Single ejecutada por la hebra 0
b[0] = 10    b[1] = 10    b[2] = 10    b[3] = 10    b[4] = 10    b[5] = 10    b[6] = 10    b[7] = 10    b[8] = 10
antoniojrr@antoniojrr-laptop:~/DGIIM/AC/bp1$

```

RESPUESTA A LA PREGUNTA:

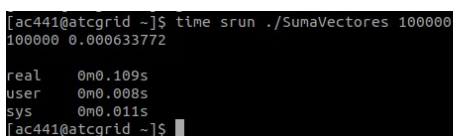
Se diferencian en que en este caso la ejecución de los *printf()* siempre será llevada a cabo por el *thread master*, con ID 0.

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: Porque si no establecemos que se espere en ese punto a que todos los threads hayan ejecutado el código previo a este, no podemos asegurar que todas las sumas se hayan realizado, de forma que si hay algún *thread* que no la haya ejecutado en el momento que la *master* llegue a imprimir por pantalla, esta no será tomada en cuenta.

3 Parte II. 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:


```

[ac441@atcgrid ~]$ time srun ./SumaVectores 100000
100000 0.000633772

real    0m0.109s
user    0m0.008s
sys     0m0.011s
[ac441@atcgrid ~]$

```

RESPUESTA: Es menor. Esto se debe a un tiempo de E/S implícito al programa

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):

```
ac441@atcgrid ~]$ gcc -O2 SumaVectores.c -o SumaVectores -lrt
ac441@atcgrid ~]$ gcc -O2 -S SumaVectores.c -lrt
ac441@atcgrid ~]$
```

```
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
.L7:
leaq    16(%rsp), %rsi
xorl    %edi, %edi
call    clock_gettime
```

```
ac441@atcgrid ~]$ srunk ./SumaVectores 10
10 0.000391151
ac441@atcgrid ~]$ srunk ./SumaVectores 10000000
10000000 0.040779032
ac441@atcgrid ~]$
```

RESPUESTA: cálculo de los MIPS y los MFLOPS

Vemos que en el primer caso se ejecutarán $(10 \cdot 6) + 5$ instrucciones, de las cuales hay $10 \cdot 1$ operaciones con coma flotante. En el segundo, se ejecutarán $(10^7 \cdot 6) + 5$ instrucciones, de las cuales hay $10^7 \cdot 1$ operaciones con coma flotante.

Vemos entonces que:

$$\text{MIPS} = ((10 \cdot 6) + 5) / (10^6 \cdot 0.000391151) = 0.16617623373$$

$$\text{MFLOPS} = (10) / (10^6 \cdot 0.000391151) = 0.02556557442$$

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

```
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
.L7:
leaq    16(%rsp), %rsi
xorl    %edi, %edi
call    clock_gettime
```

- 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`

```
//Inicializar vectores
#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<N; i++){
        v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
    }
}

int num_threads;
cgt1 = omp_get_wtime();

//Calcular suma de vectores e imprimir la suma de cada componente si N es pequeño
#pragma omp parallel
{
    num_threads = omp_get_num_threads();
    #pragma omp for
    for(i=0; i<N; i++) {
        v3[i] = v1[i] + v2[i];
    }
    if(N<20)
        printf ("v3[%i] = v1[%i] + v2[%i] = %d + %d = %d",i,i,i,v1[i],v2[i],v3[i]);
}

cgt2 = omp_get_wtime();
ncgt = cgt2 - cgt1;

//Imprimir el tiempo de ejecución
printf("%u %11.9f\t\n",N,ncgt);
```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

```
[ac441@atcgrid ~]$ ./sp-OpenMP-for 8
v3[0] = v1[0] + v2[0] = 0.800000 + 0.800000 = 1.600000
v3[6] = v1[6] + v2[6] = 1.400000 + 0.200000 = 1.600000
v3[3] = v1[3] + v2[3] = 1.100000 + 0.500000 = 1.600000
v3[4] = v1[4] + v2[4] = 1.200000 + 0.400000 = 1.600000
v3[2] = v1[2] + v2[2] = 1.000000 + 0.600000 = 1.600000
v3[5] = v1[5] + v2[5] = 1.300000 + 0.300000 = 1.600000
v3[7] = v1[7] + v2[7] = 1.500000 + 0.100000 = 1.600000
v3[1] = v1[1] + v2[1] = 0.900000 + 0.700000 = 1.600000
8 0.001536179
[ac441@atcgrid ~]$ ./sp-OpenMP-for 11
v3[7] = v1[7] + v2[7] = 1.800000 + 0.400000 = 2.200000
v3[8] = v1[8] + v2[8] = 1.900000 + 0.300000 = 2.200000
v3[4] = v1[4] + v2[4] = 1.500000 + 0.700000 = 2.200000
v3[5] = v1[5] + v2[5] = 1.600000 + 0.600000 = 2.200000
v3[6] = v1[6] + v2[6] = 1.700000 + 0.500000 = 2.200000
v3[9] = v1[9] + v2[9] = 2.000000 + 0.200000 = 2.200000
v3[10] = v1[10] + v2[10] = 2.100000 + 0.100000 = 2.200000
v3[2] = v1[2] + v2[2] = 1.300000 + 0.900000 = 2.200000
v3[3] = v1[3] + v2[3] = 1.400000 + 0.800000 = 2.200000
v3[0] = v1[0] + v2[0] = 1.100000 + 1.100000 = 2.200000
v3[1] = v1[1] + v2[1] = 1.200000 + 1.000000 = 2.200000
11 0.007524077
```

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, v3, 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 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-sections.c`

```
//Inicializar vectores
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        for(i1=0; i1<N; i1+=5){
            v1[i1] = N*0.1+i1*0.1;
            v2[i1] = N*0.1-i1*0.1;
        }

        #pragma omp section
        for(i2=1; i2<N; i2+=5){
            v1[i2] = N*0.1+i2*0.1;
            v2[i2] = N*0.1-i2*0.1;
        }

        #pragma omp section
        for(i3=2; i3<N; i3+=5){
            v1[i3] = N*0.1+i3*0.1;
            v2[i3] = N*0.1-i3*0.1;
        }

        #pragma omp section
        for(i4=3; i4<N; i4+=5){
            v1[i4] = N*0.1+i4*0.1;
            v2[i4] = N*0.1-i4*0.1;
        }

        #pragma omp section
        for(i5=4; i5<N; i5+=5){
            v1[i5] = N*0.1+i5*0.1;
            v2[i5] = N*0.1-i5*0.1;
        }
    }
}

int num_threads;
cgt1 = omp_get_wtime();

//Calcular suma de vectores
#pragma omp parallel
{
    num_threads = omp_get_num_threads();
    #pragma omp sections
    {
        #pragma omp section
        for(i1=0; i1<N; i1+=3)
            v3[i1] = v1[i1] + v2[i1];

        #pragma omp section
        for(i2=1; i2<N; i2+=3)
            v3[i2] = v1[i2] + v2[i2];
    }
}
```

```

#pragma omp section
    for(i3=2; i3<N; i3+=3)
        v3[i3] = v1[i3] + v2[i3];

#pragma omp section
    for(i4=3; i4<N; i4+=3)
        v3[i4] = v1[i4] + v2[i4];

#pragma omp section
    for(i5=4; i5<N; i5+=3)
        v3[i5] = v1[i5] + v2[i5];
}

cgt2 = omp_get_wtime();
ncgt = cgt2 - cgt1;

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

if (N<20) {
    for (i1=0; i1<N; i1++)
        printf ("\tv3[%i] = v1[%i] + v2[%i] = %f + %f = %f\n", i1, i1, i1, v1[i1], v2[i1], v3[i1]);
}
printf("%u %11.9f\t\t\n", N, ncgt);

```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

```

[ac441@atcgrid ~]$ ./sp-OpenMP-sections 8
v3[0] = v1[0] + v2[0] = 0.800000 + 0.800000 = 1.600000
v3[1] = v1[1] + v2[1] = 0.900000 + 0.700000 = 1.600000
v3[2] = v1[2] + v2[2] = 1.000000 + 0.600000 = 1.600000
v3[3] = v1[3] + v2[3] = 1.100000 + 0.500000 = 1.600000
v3[4] = v1[4] + v2[4] = 1.200000 + 0.400000 = 1.600000
v3[5] = v1[5] + v2[5] = 1.300000 + 0.300000 = 1.600000
v3[6] = v1[6] + v2[6] = 1.400000 + 0.200000 = 1.600000
v3[7] = v1[7] + v2[7] = 1.500000 + 0.100000 = 1.600000
8 0.000447452
[ac441@atcgrid ~]$ ./sp-OpenMP-sections 11
v3[0] = v1[0] + v2[0] = 1.100000 + 1.100000 = 2.200000
v3[1] = v1[1] + v2[1] = 1.200000 + 1.000000 = 2.200000
v3[2] = v1[2] + v2[2] = 1.300000 + 0.900000 = 2.200000
v3[3] = v1[3] + v2[3] = 1.400000 + 0.800000 = 2.200000
v3[4] = v1[4] + v2[4] = 1.500000 + 0.700000 = 2.200000
v3[5] = v1[5] + v2[5] = 1.600000 + 0.600000 = 2.200000
v3[6] = v1[6] + v2[6] = 1.700000 + 0.500000 = 2.200000
v3[7] = v1[7] + v2[7] = 1.800000 + 0.400000 = 2.200000
v3[8] = v1[8] + v2[8] = 1.900000 + 0.300000 = 2.200000
v3[9] = v1[9] + v2[9] = 2.000000 + 0.200000 = 2.200000
v3[10] = v1[10] + v2[10] = 2.100000 + 0.100000 = 2.200000
11 0.000396546

```

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 cuántos 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: Teniendo en cuenta que a la hora de la ejecución no se haya incluido la opción `-hint=nomulti-thread` tenemos que en el ejercicio 7 se podría utilizar N threads y $N/2$ cores, siendo N el tamaño del vector, y en el 8 se pueden utilizar como máximo 5 threads o 3 cores ya que son el número de section incluidos.

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

```
#!/bin/bash
#Órdenes para el Gestor de carga de trabajo (no intercalar instrucciones del scrip)
#1. Asignar al trabajo un nombre
#SBATCH --job-name=sp-OpenMP
#2. Asignar el trabajo a una partición (cola)
#SBATCH --partition=ac
#3. Asignar el trabajo a un account
#SBATCH --account=ac
#4. Para que el trabajo no comparta recursos
#SBATCH --exclusive
#5. Para que se genere un único proceso del sistema operativo que pueda usar un máximo de 12 núcleos
#SBATCH --ntasks 1 --cpus-per-task 12
#Se pueden añadir más órdenes para el gestor de colas

#Obtener información de las variables del entorno del Gestor de carga de trabajo:
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"

for (( CONTADOR=16384; CONTADOR<=67108864; CONTADOR*=2)) ; do

    srun ./SumaVectores $CONTADOR >> salida_secuencial.dat
    srun ./sp-OpenMP-for $CONTADOR >> salida_for.dat
    srun ./sp-OpenMP-sections $CONTADOR >> salida_sections.dat

done
```

(RECUERDE ADJUNTAR LOS CÓDIGOS AL .ZIP)

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

```
[ac441@atcgrid ~]$ sbatch sp-OpenMP-script10.sh
Submitted batch job 180358
[ac441@atcgrid ~]$
```


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.

Nº de Componentes	T. secuencial vect. Globales 1 thread = cores lógicos = cores físicos	T. paralelo (versión for) 12threads = cores lógicos = cores físicos	T. paralelo (versión sections) 12threads = cores lógicos = cores físicos
16384	0.000430144	0.001111817	0.000542618
32768	0.000406448	0.001108587	0.000794966
65536	0.000623655	0.001298979	0.001019552
131072	0.000708216	0.001457505	0.001239210
262144	0.001486103	0.002224877	0.002033744
524288	0.002421338	0.002405159	0.005008619
1048576	0.004908437	0.003566556	0.006257910
2097152	0.008970976	0.005457487	0.011801954
4194304	0.018202759	0.009225540	0.024395641
8388608	0.034679065	0.017549872	0.065500502
16777216	0.066245130	0.031249475	0.123922758
33554432	0.131346279	0.062117703	0.147597712
67108864	0.262193565	0.122528960	0.228628326

Nº de Componentes	T. secuencial vect. Globales 1 thread = cores lógicos = cores físicos MIPC	T. paralelo (versión for) 4 threads = cores lógicos = cores físicos MIPC	T. paralelo (versión sections) 4 threads = cores lógicos = cores físicos MIPC
16384	0.000112031	0.000047526	0.000163495
32768	0.000186463	0.000035464	0.000162826
65536	0.000243316	0.000058839	0.000211280
131072	0.000407365	0.000098192	0.000372498
262144	0.000642899	0.000137635	0.000674587
524288	0.001250482	0.000437700	0.001445157
1048576	0.003081476	0.001090099	0.003096725
2097152	0.004808393	0.002496618	0.007980025
4194304	0.009736467	0.004128157	0.012058271
8388608	0.020023207	0.007555535	0.025179557
16777216	0.037058448	0.014819167	0.046158172
33554432	0.087054922	0.030627672	0.090513653
67108864	0.154685289	0.053726106	0.179548796

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 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`

```
#!/bin/bash
#Órdenes para el Gestor de carga de trabajo (no intercalar instrucciones del scrip
#1. Asignar al trabajo un nombre
#SBATCH --job-name=helloOMP
#2. Asignar el trabajo a una partición (cola)
#SBATCH --partition=ac
#3. Asignar el trabajo a un account
#SBATCH --account=ac
#4. Para que el trabajo no comparta recursos
#SBATCH --exclusive
#5. Para que se genere un único proceso del sistema operativo que pueda usar un máximo de 12 núcleos
#SBATCH --ntasks 1 --cpus-per-task 12
#Se pueden añadir más órdenes para el gestor de colas

#Obtener información de las variables del entorno del Gestor de carga de trabajo:
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"

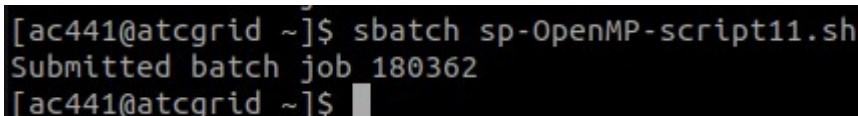
for (( CONTADOR=8388608; CONTADOR<=67108864; CONTADOR*=2)) ; do

    (time srun ./SumaVectores $CONTADOR) >> salida2_secuencial.dat
    (time srun ./sp-OpenMP-for $CONTADOR) >> salida2_for.dat

done
```

(RECUERDE ADJUNTAR LOS CÓDIGOS AL .ZIP)

CAPTURAS DE PANTALLA (ejecución en atcgrid):



```
[ac441@atcgrid ~]$ sbatch sp-OpenMP-script11.sh
Submitted batch job 180362
[ac441@atcgrid ~]$
```

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	0.036966398	0.009	0.012	0.029237207	0.009	0.011
16777216	0.067005674	0.014	0.005	0.055463031	0.014	0.007
33554432	0.131978531	0.009	0.008	0.108667981	0.009	0.008
67108864	0.132627832	0.009	0.007	0.107840870	0.008	0.011