

2º curso / 2º cuatr.
Grados Ing. In-
form.

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`

```
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
    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 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`

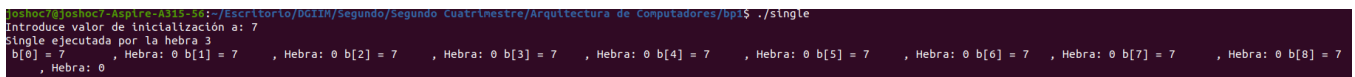
CAPTURAS DE PANTALLA:

```
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("%d\t", b[i]);
    printf("\n");
}
return(0);
}
```



```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DCIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp1$ ./single
Introduce valor de inicialización a: 7
Single ejecutada por la hebra 3
b[0] = 7, Hebra: 0 b[1] = 7, Hebra: 0 b[2] = 7, Hebra: 0 b[3] = 7, Hebra: 0 b[4] = 7, Hebra: 0 b[5] = 7, Hebra: 0 b[6] = 7, Hebra: 0 b[7] = 7, Hebra: 0 b[8] = 7, Hebra: 0
```

- 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`

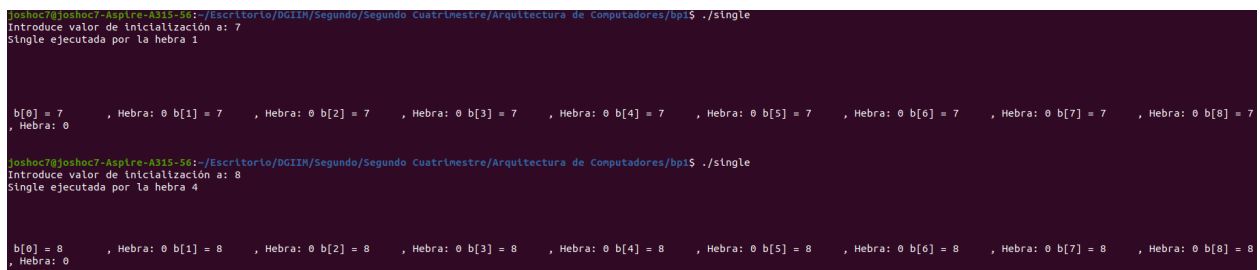
CAPTURAS DE PANTALLA:

```
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
    for (i=0; i<n; i++)
        printf("%d\t", b[i]);
    printf("\n");
}
return(0);
}
```



```
joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DCIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp1$ ./single
Introduce valor de inicialización a: 7
Single ejecutada por la hebra 1

b[0] = 7, Hebra: 0 b[1] = 7, Hebra: 0 b[2] = 7, Hebra: 0 b[3] = 7, Hebra: 0 b[4] = 7, Hebra: 0 b[5] = 7, Hebra: 0 b[6] = 7, Hebra: 0 b[7] = 7, Hebra: 0 b[8] = 7, Hebra: 0

joshoc7@joshoc7-Aspire-A315-56:~/Escritorio/DCIIM/Segundo/Segundo Cuatrimestre/Arquitectura de Computadores/bp1$ ./single
Introduce valor de inicialización a: 8
Single ejecutada por la hebra 4

b[0] = 8, Hebra: 0 b[1] = 8, Hebra: 0 b[2] = 8, Hebra: 0 b[3] = 8, Hebra: 0 b[4] = 8, Hebra: 0 b[5] = 8, Hebra: 0 b[6] = 8, Hebra: 0 b[7] = 8, Hebra: 0 b[8] = 8, Hebra: 0
```

RESPUESTA A LA PREGUNTA: La diferencia con respecto a la ejecución en el anterior ejercicio es que la impresión de resultados siempre es realizada por la hebra 0 (la directiva master obliga a que así sea).

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 se debe a que con la directiva `barrier` obligamos a que los threads se esperen entre sí y de esta forma nos aseguramos que la variable privada de cada thread es sumada a la variable global suma. Si eliminamos `barrier`, podría darse que suma se imprima antes de que todos los threads hayan calculado su `sumalocal` y la hayan añadido a suma, dando lugar a un valor final de suma erróneo.

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:

```
[ac412@atcgrid ~]$ time srun ./SumaVectores 10000000
Tamaño Vectores:10000000 (4 B)
Tiempo:0.038136447 / Tamaño Vectores:10000000 / V1[0]+V2[0]=V3[0](1000
000.000000+1000000.000000=2000000.000000) / / V1[9999999]+V2[9999999]=V3[9999999
](1999999.900000+0.100000=2000000.000000) /
real    0m0.195s
user    0m0.007s
sys     0m0.008s
```

RESPUESTA: La suma de los tiempos de usuario y de sistema es menor que el tiempo real, esto se debe a que hay un tiempo de espera por E/S (entradas y salidas) implícito.

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): Para generar el código ensamblador he usado el orden `gcc -O2 -S SumaVectores.c -lrt` y para el código ejecutable `gcc -O2 SumaVectores.c -o SumaVectores -lrt`. Mostramos a continuación la ejecución para 10 y 100000:

```
call    clock_gettime
xorl    %eax, %eax
.p2align 4,,10
.p2align 3

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
```

```
[ac412@atcgrid ~]$ srun ./SumaVectores 10
Tamaño Vectores:10 (4 B)
Tiempo:0.000377979 / 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) /
[ac412@atcgrid ~]$ srun ./SumaVectores 10000000
Tamaño Vectores:10000000 (4 B)
Tiempo:0.041159948 / 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) /
```

RESPUESTA: cálculo de los MIPS y los MFLOPS

Empecemos con el cálculo de los MIPS y MFLOPS para la ejecución de SumaVectores para el tamaño 10:

MIPS = $(6 \text{ instrucciones} \times 10 \text{ iteraciones} + 5) / (0.000377979 \times 10^6) \text{ s} = 0.17197 \text{ MIPS}$

MFLOPS = $(1 \text{ instrucciones} \times 10 \text{ iteraciones} + 5) / (0.000377979 \times 10^6) = 0.03968 \text{ MFLOPS}$

Y análogamente para cuando ejecutamos SumaVectores para el tamaño 10000000:

MIPS = $(6 \text{ instrucciones} \times 10000000 \text{ iteraciones} + 5) / (0.0411599648 \times 10^6) \text{ s} = 1457.73 \text{ MIPS}$

MFLOPS = $(1 \text{ instrucciones} \times 10000000 \text{ iteraciones} + 5) / (0.0411599648 \times 10^6) = 242.9547 \text{ MFLOPS}$

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

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`

```
//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

#pragma omp parallel
{
    num_threads = omp_get_num_threads();
    #pragma omp for
    for(i=0; i<N; i++){
        v3[i] = v1[i] + v2[i];
    }
}

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

//Imprimir resultado de la suma y el tiempo de ejecución
if (N<10) {
    printf("Tiempo:%11.9f\t / Tamaño Vectores:%u / Número de threads: %d\n",ncgt,N,num_threads);
    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 / Número de hebras: %d / 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,num_threads,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
```

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

```
[ac412@atcgrid ~]$ gcc -O2 -fopenmp -o sp-OpenMP-for sp-OpenMP-for.c -lrt
[ac412@atcgrid ~]$ srun ./sp-OpenMP-for 8
Tamaño Vectores:8 (4 B)
Tiempo:0.000531381 / Tamaño Vectores:8 / Número de threads: 2
/ 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) /
[ac412@atcgrid ~]$ srun ./sp-OpenMP-for 11
Tamaño Vectores:11 (4 B)
Tiempo:0.000535393 / Tamaño Vectores:11 / Número de hebras: 2 / V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000) / / V1[10]+V2[10]=V3[10](2.100000+0.100000=2.200000)
```

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(i=0; i<N; i+=3){
            v1[i] = N*0.1+i*0.1;
            v2[i] = N*0.1-i*0.1;
        }

        #pragma omp section
        for(i=1; i<N; i+=3){
            v1[i] = N*0.1+i*0.1;
            v2[i] = N*0.1-i*0.1;
        }

        #pragma omp section
        for(i=2; i<N; i+=3){
            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
#pragma omp parallel
{
    num_threads = omp_get_num_threads();
    #pragma omp sections
    {
        #pragma omp section
        for(i=0; i<N; i+=3){
            v3[i] = v1[i] + v2[i];
        }

        #pragma omp section
        for(i=1; i<N; i+=3){
            v2[i] = v1[i] + v2[i];
        }

        #pragma omp section
        for(i=2; i<N; i+=3){
            v3[i] = v1[i] + v2[i];
        }
    }
}
```

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

```
[ac412@atcgrid ~]$ srun -c3 ./sp-OpenMP-sections 8
Tamaño Vectores:8 (4 B)
Tiempo:0.000485274 / Tamaño Vectores:8 / Número de threads: 4
/ 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) /
[ac412@atcgrid ~]$ srun -c3 ./sp-OpenMP-sections 11
Tamaño Vectores:11 (4 B)
Tiempo:0.000576742 / Tamaño Vectores:11 / Número de hebras: 4 / V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000) / / V1[10]+V2[10]=V3[10](2.100000+0.100000=2.200000)
```

9. ¿Cuántos threads y cuántos cores como máximo podría utilizar la versión que ha implementado en el ejercicio 7? 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: Aunque podemos usar tantos threads y cores como soporte nuestro PC, lo más natural en el programa del ejercicio 7 es que podamos usar tantos threads como número de iteraciones haya. Por ejemplo, para la ejecución con tamaño $N = 8$, podremos usar hasta 8 threads como máximo, encargándose cada uno de una iteración. En el ejercicio 8, podremos usar tantos threads como sections hayamos delimitado (en nuestro caso, 4).

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
#!/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=65536; CONTADOR<=67108864; CONTADOR*=2)); do
    srun ./SumaVectores $CONTADOR >> salida.dat
done

for (( CONTADOR=65536; CONTADOR<=67108864; CONTADOR*=2)); do
    srun ./sp-OpenMP-for $CONTADOR >> salida_for.dat
done

for (( CONTADOR=65536; CONTADOR<=67108864; CONTADOR*=2)); do
    srun ./sp-OpenMP-sections $CONTADOR >> salida_sections.dat
done
```

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

```
[ac412@atcgrid ~]$ sbatch ./script.sh
Submitted batch job 138810
[ac412@atcgrid ~]$ cat slurm-138810.out
Id. usuario del trabajo: ac412
Id. del trabajo: 138810
Nombre del trabajo especificado por usuario: helloOMP
Directorio de trabajo (en el que se ejecuta el script): /home/ac412
Cola: ac
Nodo que ejecuta este trabajo:atcgrid.ugr.es
Nº de nodos asignados al trabajo: 1
Nodos asignados al trabajo: atcgrid1
CPUs por nodo: 24
```

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.

Como mi portátil se ha puesto rebelde y no me deja usar LibreOfficeCalc, pondré los datasets. Se usan 8 threads en los códigos en paralelo en mi PC

Secuencial

```
65536 0.000487175
131072 0.000878070
262144 0.000828405
524288 0.003524371
1048576 0.004076108
2097152 0.006544645
4194304 0.012873764
8388608 0.026143802
16777216 0.049876008
```

Paralelo for

```
65536 0.000087033
131072 0.000255095
262144 0.000369434
524288 0.001046953
1048576 0.001820878
2097152 0.003811881
4194304 0.027350669
8388608 0.014773071
16777216 0.027066652
```

Paralelo sections

```
65536 0.000306088
131072 0.000598309
262144 0.001305303
524288 0.002573881
1048576 0.007519817
2097152 0.024479155
4194304 0.028445687
8388608 0.035541553
16777216 0.071055210
```

Y en el atcgrid (24 threads como se puede ver en la captura anterior:

Secuencial

```
65536 0.000528239
131072 0.000700354
262144 0.001412598
524288 0.002730427
```

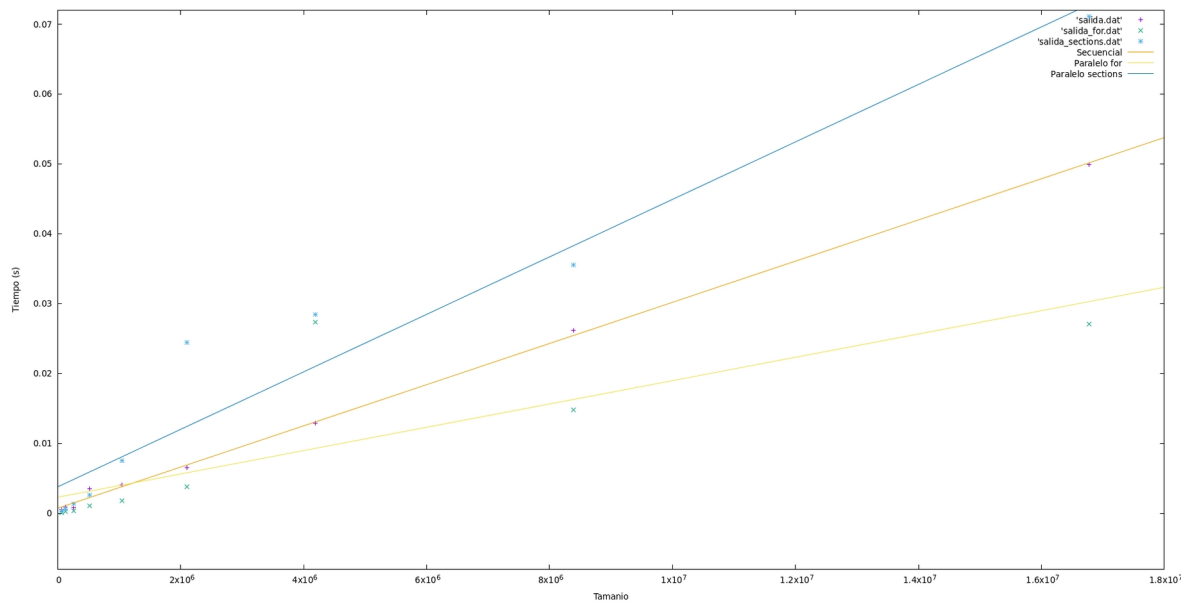
1048576 0.005177045
2097152 0.009744345
4194304 0.018062761
8388608 0.033999789
16777216 0.066968294

Paralelo for

65536 0.002933573
131072 0.003239726
262144 0.003939217
524288 0.004869669
1048576 0.006260240
2097152 0.008688337
4194304 0.014982947
8388608 0.029416184
16777216 0.057384295

Paralelo sections

65536 0.004220100
131072 0.000880590
262144 0.001794425
524288 0.003099743
1048576 0.006542645
2097152 0.010545948
4194304 0.020750208
8388608 0.040995548
16777216 0.079851238



La gráfica para los tiempos del atcgrid no la hago porque es análoga.

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`

Explicaré este ejercicio sin hacer lo que se pide. El script es muy análogo al del anterior ejercicio solo que deberemos usar la orden `time` delante del `./ejecutable`, siendo ejecutable el programa cuyos tiempos queremos obtener. En prácticas de años pasados, en la ejecución del código en paralelo normalmente el tiempo de CPU era mayor que el real. Esto se debe al tiempo que se invierte en crear y destruir hebras.