

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): Valentino Lugli

Grupo de prácticas y profesor de prácticas: C1, Christian Morillas

Fecha de entrega: 12/04/2021

Fecha evaluación en clase: 13/04/2021

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`

```
int main(int argc, char **argv)
{
    int i, n = 9;
    if(argc < 2)
    {
        fprintf(stderr, "\n[ERROR] - Falta no iteraciones \n");
        exit(-1);
    }

    n = atoi(argv[1]);
    #pragma omp parallel for
    {
        for (i=0; i<n; i++)
            printf("thread %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;
}
```

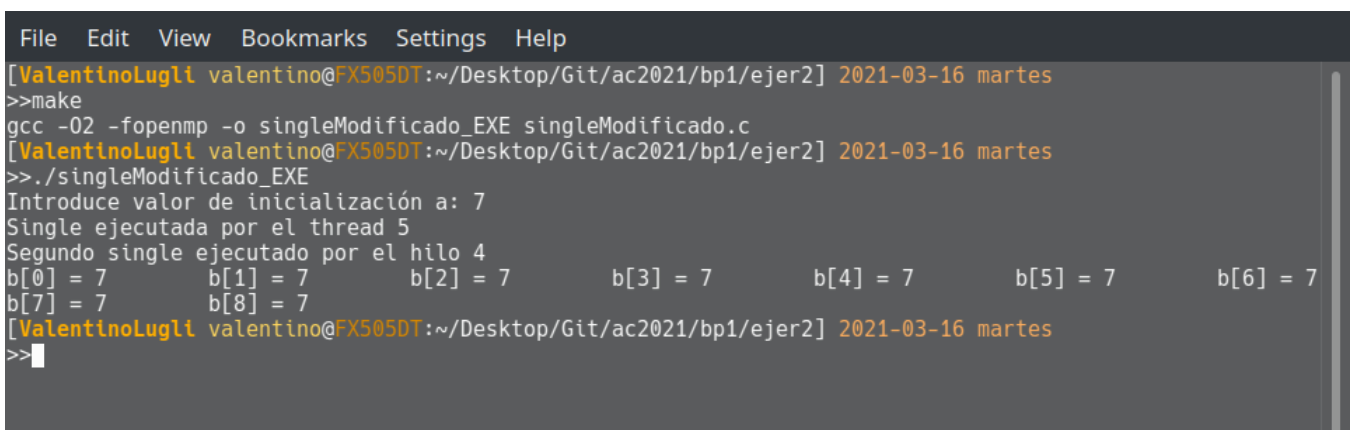
2. Imprimir los resultados del programa `single.c` usando una directiva `single` dentro de la construcción `parallel` en lugar de imprimirlos fuera de la región `parallel`. Añadir lo necesario, dentro de la nueva directiva `single` incorporada, para que se imprima el identificador del thread que ejecuta 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`

```
int main()
{
    int n = 9, 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 el thread %d\n", omp_get_thread_num());
        }
        #pragma omp for
        for (i=0; i<n; i++)
            b[i] = a;

        #pragma omp single
        {
            printf("Segundo single ejecutado por el hilo %d\n",
omp_get_thread_num());
            for (i=0; i<n; i++)
                printf("b[%d] = %d\t", i, b[i]);
            printf("\n");
        }
    }
}
```

CAPTURAS DE PANTALLA:



```
File Edit View Bookmarks Settings Help
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer2] 2021-03-16 martes
>>make
gcc -O2 -fopenmp -o singleModificado_EXE singleModificado.c
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer2] 2021-03-16 martes
>>./singleModificado_EXE
Introduce valor de inicialización a: 7
Single ejecutada por el thread 5
Segundo single ejecutado por el hilo 4
b[0] = 7      b[1] = 7      b[2] = 7      b[3] = 7      b[4] = 7      b[5] = 7      b[6] = 7
b[7] = 7      b[8] = 7
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer2] 2021-03-16 martes
>>
```

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, 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 el thread %d\n", omp_get_thread_num());
        }
        #pragma omp for
        for (i=0; i<n; i++)
            b[i] = a;

        #pragma omp master
        {
            printf("Segundo single ejecutado por el hilo %d\n",
omp_get_thread_num());
            for (i=0; i<n; i++)
                printf("b[%d] = %d\t", i, b[i]);
            printf("\n");
        }
    }
}
```

CAPTURAS DE PANTALLA:

```
File Edit View Bookmarks Settings Help
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer3] 2021-03-16 martes
>>make
gcc -O2 -fopenmp -o singleModificado2_EXE singleModificado2.c
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer3] 2021-03-16 martes
>>./singleModificado2_EXE
Introduce valor de inicialización a: 16
Single ejecutada por el thread 3
Segundo single ejecutado por el hilo 0
b[0] = 16      b[1] = 16      b[2] = 16      b[3] = 16      b[4] = 16      b[5] = 16      b[6] = 1
6      b[7] = 16      b[8] = 16
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer3] 2021-03-16 martes
>>|
```

RESPUESTA A LA PREGUNTA:

La diferencia que se observa en este programa es que la hebra que imprime el mensaje es la hebra maestra, la cual es siempre la número cero. Si se utiliza la directiva `single`, puede ser cualquier hebra quien imprima el mensaje; en este caso no hay más diferencias aunque la directiva `master` a diferencia de `single` no posee barrera implícita, por lo que si hubiera algún más código debajo esta directiva sería ejecutado por las hebras sin que estas esperasen por que finalizara la ejecución de hebra master de la directiva.

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:

La suma no será correcta siempre debido a que, primero que todo, la directiva `atomic` no posee barreras implícitas, lo que quiere decir que una hebra ejecuta la operación atómica y luego continúa ejecutando el código

sin esperar al resto. Las hebras que no son maestras al encontrarse luego con la directiva `master` simplemente finalizan su ejecución porque el código finaliza justamente después. Cuando la hebra maestra ejecuta el código dentro de la directiva `master` puede que ya se hayan ejecutado el resto de hebras o puede que no, de ahí proviene la discrepancia de las sumas.

1.1.1

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:

```
File Edit View Bookmarks Settings Help
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer5] 2021-03-16 martes
>> sbatch -pac -Aac -n1 -c1 --hint=nomultithread --exclusive --wrap "time ./SumaVectores_seq_EXE 10000000"
Submitted batch job 72519
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer5] 2021-03-16 martes
>> cat slurm-72519.out
Tiempo:0.036608205 / Tamaño Vectores:10000000 / V1[0]+V2[0]=V3[0](2.308568+1.591894=3.900462)
/ / V1[9999999]+V2[9999999]=V3[9999999](0.368681+0.626338=0.995020) /

real    0m0.604s
user    0m0.538s
sys     0m0.048s
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer5] 2021-03-16 martes
>>
```

RESPUESTA:

El tiempo CPU, siendo este el tiempo de usuario (`user`) más el sistema (`sys`) es $0.538 + 0.048 = 0.586$ s, esto es menor que el tiempo real de ejecución (*elapsed* o también *real*) de 0.604s. Esto es así porque el tiempo que se tiene añadido es por las E/S asociadas a otros programas en ejecución en el nodo atcgrid.

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. 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):

```
File Edit View Bookmarks Settings Help
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer5] 2021-03-23 martes
>> make
gcc -O2 -o SumaVectores_seq_EXE SumaVectores.c
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer5] 2021-03-23 martes
>> make SumaVectores.s
gcc -O2 -S SumaVectores.c
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer5] 2021-03-23 martes
>>
```

```

File Edit View Bookmarks Settings Help
[ValentinoLugli ciestudiante16@atcgrid:~/bp1/ejer5] 2021-03-23 martes
>>srun -pac -Aac -n1 -c1 --hint=nomultithread --exclusive ./SumaVectores_seq_EXE 10
Tiempo:0.000000162 / Tamaño Vectores:10 / V1[0]+V2[0]=V3[0](0.339999+2.278424=2.618422) / / V1[9]
]+V2[9]=V3[9](1.070797+47.456381=48.527178) /
[ValentinoLugli ciestudiante16@atcgrid:~/bp1/ejer5] 2021-03-23 martes
>>srun -pac -Aac -n1 -c1 --hint=nomultithread --exclusive ./SumaVectores_seq_EXE 100000000
Tiempo:0.037207039 / Tamaño Vectores:100000000 / V1[0]+V2[0]=V3[0](1.187767+1.001317=2.189085)
/ / V1[99999999]+V2[99999999]=V3[99999999](0.563268+1.315458=1.878726) /
[ValentinoLugli ciestudiante16@atcgrid:~/bp1/ejer5] 2021-03-23 martes
>>

```

RESPUESTA: cálculo de los MIPS y los MFLOPS

La fórmula para MIPS es $MIPS = \frac{NI}{T_{CPU} \times 10^6}$; de aquí NI es el número de instrucciones que se obtienen contando las instrucciones en ensamblador, entre las llamadas de `clock_gettime`. Hay 3 instrucciones fuera del bucle y 6 instrucciones dentro del bucle. El T_{CPU} se obtiene del tiempo que produce el programa, por lo tanto queda que

$$MIPS = \frac{3 + (10 \cdot 6)}{(1.62 \times 10^{-7}) \times 10^6} = 388.8889$$

$$MIPS = \frac{3 + (100000000 \cdot 6)}{0.037207039 \times 10^6} = 1612.5982$$

Con respecto a los MFLOPS, la fórmula es $MFLOPS = \frac{OpsComaFlotante}{T_{CPU} \times 10^6}$. Las operaciones en coma flotante son 3, aquellas que utilizan el registro `%xmm0` dentro del bucle en el código, el tiempo es el mismo utilizado para obtener los MIPS.

$$MFLOPS = \frac{3 \times 10}{(1.62 \times 10^{-7}) \times 10^6} = 185.1852$$

$$MFLOPS = \frac{3 \times 100000000}{0.037207039 \times 10^6} = 806.299$$

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

```

movsd    %xmm7, (%r14,%rax,8)
addq     $1, %rax
cmpl     %eax, %r12d
ja       .L7
.L8:
leaq     16(%rsp), %rsi
xorl     %edi, %edi
call     clock_gettime@PLT
xorl     %eax, %eax
.p2align 4,,10
.p2align 3
.L10:
movsd    0(%rbp,%rax,8), %xmm0
addsd    (%r14,%rax,8), %xmm0
movsd    %xmm0, 0(%r13,%rax,8)
addq     $1, %rax
cmpl     %eax, %r12d
ja       .L10

```

```

leaq    32(%rsp), %rsi
xorl    %edi, %edi
call    clock_gettime@PLT
movq    40(%rsp), %rax
pxor    %xmm0, %xmm0
subq    24(%rsp), %rax

```

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 `sp-OpenMP-for.c`

```

int main(int argc, char** argv){

    int i;
    double start;
    double finish;

    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2){
        printf("Faltan nº componentes del vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)
    //printf("Tamaño Vectores:%u (%u B)\n",N, sizeof(unsigned int));

    double *v1, *v2, *v3;
    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) || (v2 == NULL)) {
        printf("No hay suficiente espacio para los vectores \n");
        exit(-2);
    }

    //Inicializar vectores
    if (N < 9)
    {
        #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;
        }
    }
}

```

```

else
{
    srand(time(0));

    #pragma omp parallel
    {
        unsigned int myseed = omp_get_thread_num();
        #pragma omp for
        for (i = 0; i < N; i++)
        {
            v1[i] = rand_r(&myseed)/ ((double) rand_r(&myseed));
            v2[i] = rand_r(&myseed)/ ((double) rand_r(&myseed)); //printf("%d:%f,
%f/", i, v1[i], v2[i]);
        }
    }

    start = omp_get_wtime();

    //Calcular suma de vectores
    #pragma omp parallel for
    for(i=0; i<N; i++)
        v3[i] = v1[i] + v2[i];

    finish = omp_get_wtime();
    // FIN PARALELISMO

    double ncgt = finish - start;

    //Imprimir resultado de la suma y el tiempo de ejecución
    if (N<12) {
        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,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]);

    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
    free(v3); // libera el espacio reservado para v3
    return 0;
}

```


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

Nota: se cambió la función `rand()` por `rand_r()` ya que esta segunda funciona más rápido y es segura de utilizar en entornos de múltiples hilos.

```
File Edit View Bookmarks Settings Help
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer7] 2021-03-20 sábado
>>make
gcc -O2 -fopenmp -o sp-OpenMP-for_EXE sp-OpenMP-for.c
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer7] 2021-03-20 sábado
>>./sp-OpenMP-for_EXE 8
Tiempo:0.000000812 / 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) /
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/eier7] 2021-03-20 sábado
/ 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) /
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer7] 2021-03-20 sábado
>>./sp-OpenMP-for_EXE 11
Tiempo:0.000001253 / Tamaño Vectores:11
/ V1[0]+V2[0]=V3[0](0.794386+0.956818=1.751204) /
/ V1[1]+V2[1]=V3[1](1.197294+0.870400=2.067695) /
/ V1[2]+V2[2]=V3[2](2.779243+3.155433=5.934675) /
/ V1[3]+V2[3]=V3[3](0.270279+0.625956=0.896235) /
/ V1[4]+V2[4]=V3[4](1.230955+0.432234=1.663189) /
/ V1[5]+V2[5]=V3[5](0.357359+0.428790=0.786149) /
/ V1[6]+V2[6]=V3[6](213.937601+2.003599=215.941199) /
/ V1[7]+V2[7]=V3[7](0.018153+1.013517=1.031670) /
/ V1[8]+V2[8]=V3[8](0.003347+0.063940=0.067287) /
/ V1[9]+V2[9]=V3[9](2.594206+5.911152=8.505357) /
/ V1[10]+V2[10]=V3[10](2.068504+4.122500=6.191004) /
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer7] 2021-03-20 sábado
>>■
```

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); (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-sections.c`

```
int main(int argc, char** argv){

    int i;
    double start;
    double finish;

    //Leer argumento de entrada (nº de componentes del vector)
    if (argc<2){
        printf("Faltan nº componentes del vector\n");
```



```

    exit(-1);
}

    unsigned int N = atoi(argv[1]);           // Máximo N = 2^32-1=4294967295
(sizeof(unsigned int) = 4 B)
    //printf("Tamaño Vectors:%u (%u B)\n",N, sizeof(unsigned int));

double *v1, *v2, *v3;
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);
}

//Inicializar vectores
#pragma omp parallel
{
    if (N < 9)
    {
        #pragma omp sections private (i)
        {
            #pragma omp section
            {
                for (i = 0; i < N/4; i++)
                {
                    v1[i] = N * 0.1 + i * 0.1;
                    v2[i] = N * 0.1 - i * 0.1;
                }
            }

            #pragma omp section
            {
                for (i = N/4; i < (N/4)*2; i++)
                {
                    v1[i] = N * 0.1 + i * 0.1;
                    v2[i] = N * 0.1 - i * 0.1;
                }
            }

            #pragma omp section
            {
                for (i = (N/4)*2; i < (N/4)*3; i++)
                {
                    v1[i] = N * 0.1 + i * 0.1;
                    v2[i] = N * 0.1 - i * 0.1;
                }
            }

            #pragma omp section
            {
                for (i = (N/4)*3; i < N; i++)
                {
                    v1[i] = N * 0.1 + i * 0.1;
                    v2[i] = N * 0.1 - i * 0.1;
                }
            }
        }
    }
}

```

```

else
{
    unsigned int myseed = omp_get_thread_num();
    #pragma omp single
    {
        srand(time(0));
    }

    #pragma omp sections private (i)
    {
        #pragma omp section
        {
            for (i = 0; i < N/4; i++)
            {
                v1[i] = rand_r(&myseed)/ ((double) rand_r(&myseed));
                v2[i] = rand_r(&myseed)/ ((double) rand_r(&myseed)); //printf("%d:%f,
%f/", i, v1[i], v2[i]);
            }
        }

        #pragma omp section
        {
            for (i = N/4; i < (N/4)*2; i++)
            {
                v1[i] = rand_r(&myseed)/ ((double) rand_r(&myseed));
                v2[i] = rand_r(&myseed)/ ((double) rand_r(&myseed)); //printf("%d:%f,
%f/", i, v1[i], v2[i]);
            }
        }

        #pragma omp section
        {
            for (i = (N/4)*2; i < (N/4)*3; i++)
            {
                v1[i] = rand_r(&myseed)/ ((double) rand_r(&myseed));
                v2[i] = rand_r(&myseed)/ ((double) rand_r(&myseed)); //printf("%d:%f,
%f/", i, v1[i], v2[i]);
            }
        }

        #pragma omp section
        {
            for (i = (N/4)*3; i < N; i++)
            {
                v1[i] = rand_r(&myseed)/ ((double) rand_r(&myseed));
                v2[i] = rand_r(&myseed)/ ((double) rand_r(&myseed)); //printf("%d:%f,
%f/", i, v1[i], v2[i]);
            }
        }
    }

    start = omp_get_wtime();

    //Calcular suma de vectores
    #pragma omp sections private (i)
    {
        #pragma omp section
        {
            for (i = 0; i < N/4; i++)

```

```

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

    #pragma omp section
    {
        for (i = N/4; i < (N/4)*2; i++)
            v3[i] = v1[i] + v2[i];
    }

    #pragma omp section
    {
        for (i = (N/4)*2; i < (N/4)*3; i++)
            v3[i] = v1[i] + v2[i];
    }

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

finish = omp_get_wtime();
} // FIN PARALELISMO

double ncgt = finish - start;

//Imprimir resultado de la suma y el tiempo de ejecución
if (N<12) {
    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,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]);

free(v1); // libera el espacio reservado para v1
free(v2); // libera el espacio reservado para v2
free(v3); // libera el espacio reservado para v3
return 0;
}

```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

```

File Edit View Bookmarks Settings Help
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer8] 2021-03-23 martes
>>make
gcc -O2 -fopenmp -o sp-OpenMP-sections_EXE sp-OpenMP-sections.c
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer8] 2021-03-23 martes
>>./sp-OpenMP-sections_EXE 8
Tiempo:0.000840612 / 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) /
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer8] 2021-03-23 martes
>>./sp-OpenMP-sections_EXE 11
>>./sp-OpenMP-sections_EXE 11
Tiempo:0.000001542 / Tamaño Vectores:11
/ V1[0]+V2[0]=V3[0](1.192474+1.093674=2.286148) /
/ V1[1]+V2[1]=V3[1](0.413783+1.056462=1.470245) /
/ V1[2]+V2[2]=V3[2](1.088773+0.372427=1.461200) /
/ V1[3]+V2[3]=V3[3](1.011316+0.919715=1.931031) /
/ V1[4]+V2[4]=V3[4](0.305480+1.480820=1.786300) /
/ V1[5]+V2[5]=V3[5](2.896884+1.246844=4.143728) /
/ V1[6]+V2[6]=V3[6](0.567084+0.375656=0.942740) /
/ V1[7]+V2[7]=V3[7](0.175889+1.484013=1.659903) /
/ V1[8]+V2[8]=V3[8](0.080944+0.329432=0.410376) /
/ V1[9]+V2[9]=V3[9](4.073490+1.065175=5.138665) /
/ V1[10]+V2[10]=V3[10](1.037411+0.143924=1.181335) /
[ValentinoLugli valentino@FX505DT:~/Desktop/Git/ac2021/bp1/ejer8] 2021-03-23 martes
>>

```

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:

En el código ejercicio 7 se pueden utilizar en principio la cantidad de hilos y de núcleos que se necesiten, ya que OpenMP se encarga de dividir el código entre los hilos que se tienen disponibles; en cambio en el ejercicio 8 se tiene como un máximo de 4 hilos y un solo núcleo porque el código está hecho de esta manera lo cual resulta menos portable y práctico.

10. Rellenar una tabla como la Tabla 215 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
#Ordenes para el Gestor de carga de trabajo:
#1. Asigna al trabajo un nombre
#SBATCH --job-name=bp1_10
#2. Asignar el trabajo a una partición (cola)
#SBATCH --partition=ac

```

```

#3. Asignar el trabajo a un account
#SBATCH --account=ac

#Obtener informacion 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"

#Instrucciones del script para ejecutar código:

echo -e "---SECUENCIAL---"
for ((i=14;i<27;i=i+1))
do
    srun ./SumaVectores_seq_EXE $((2**$i))
done

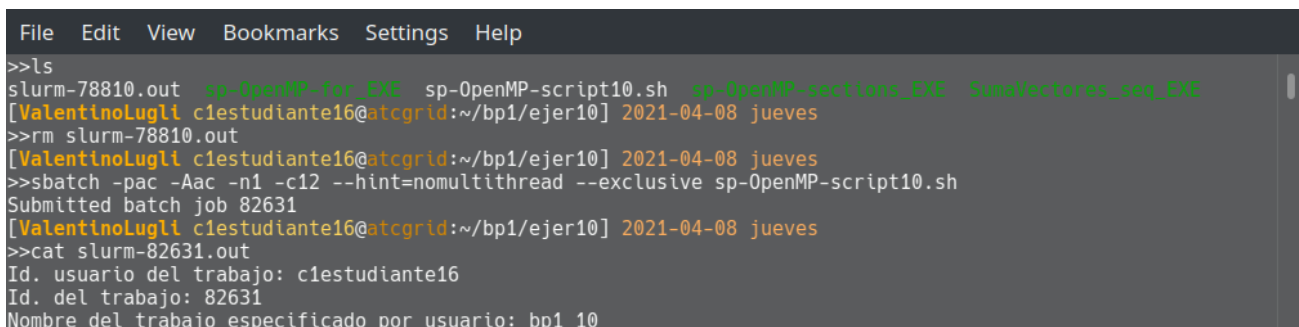
echo -e "---PARALLEL FOR---"
for ((i=14;i<27;i=i+1))
do
    srun ./sp-OpenMP-for_EXE $((2**$i))
done

echo -e "---PARALLEL SECTIONS---"
for ((i=14;i<27;i=i+1))
do
    srun ./sp-OpenMP-sections_EXE $((2**$i))
done

```

(RECUERDE ADJUNTAR LOS CÓDIGOS AL .ZIP)

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



```

File Edit View Bookmarks Settings Help
>>ls
slurm-78810.out sp-OpenMP-for_EXE sp-OpenMP-script10.sh sp-OpenMP-sections_EXE SumaVectores_seq_EXE
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer10] 2021-04-08 jueves
>>rm slurm-78810.out
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer10] 2021-04-08 jueves
>>sbatch -pac -Aac -n1 -c12 --hint=nomultithread --exclusive sp-OpenMP-script10.sh
Submitted batch job 82631
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer10] 2021-04-08 jueves
>>cat slurm-82631.out
Id. usuario del trabajo: c1estudiante16
Id. del trabajo: 82631
Nombre del trabajo especificado por usuario: bp1 10

```

Mi PC			
Nº de Componentes	T. secuencial vect. Globales	T. paralelo (versión for)	T. paralelo (versión sections)
	1 thread=core	4 threads = cores lógicos = cores físicos	4 threads = cores lógicos = cores físicos
16384	0,000044414	0,000025368	0,000042360
32768	0,000077916	0,000077034	0,000059060
65536	0,00030914	0,000131937	0,000111319
131072	0,000358733	0,000270056	0,000305142
262144	0,001052494	0,000772389	0,000484108
524288	0,00190332	0,001003732	0,001261386
1048576	0,003611814	0,002095751	0,001966969
2097152	0,006754919	0,003457685	0,003923429
4194304	0,015478111	0,007052878	0,007361567
8388608	0,029313009	0,013200409	0,013624595
16777216	0,065628995	0,028592569	0,027330692
33554432	0,095536519	0,051386621	0,057836639
67108864	0,213671175	0,101969696	0,103208119

Nodo de cómputo ATCGRID			
Nº de Componentes	T. secuencial vect. Globales	T. paralelo (versión for)	T. paralelo (versión sections)
	1 thread=core	12 threads = cores lógicos = cores físicos	12 threads = cores lógicos = cores físicos
16384	0,000118509	0,000030380	0,000026304
32768	0,000227884	0,000079330	0,000069484
65536	0,00041284	0,000090506	0,000158664
131072	0,000738466	0,000157647	0,000284534
262144	0,001874068	0,000287827	0,00056928
524288	0,002236076	0,001158901	0,001118284
1048576	0,004495096	0,003075778	0,002437755
2097152	0,008662551	0,005112547	0,003804028
4194304	0,016863262	0,007899381	0,006812926
8388608	0,032485764	0,015416734	0,02369016
16777216	0,064084317	0,029871114	0,025264394
33554432	0,126626583	0,054591686	0,069898598
67108864	0,240724358	0,114865839	0,141904466

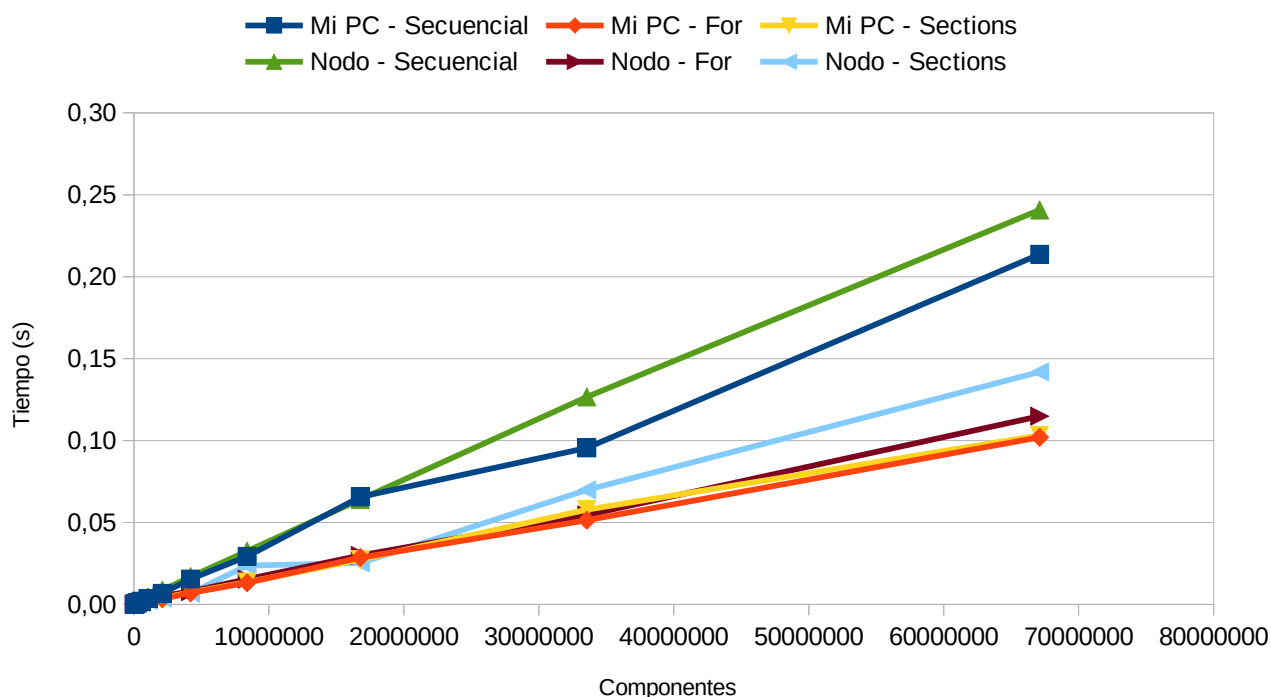


Tabla 2. Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas. Sustituir en el encabezado de la tabla “¿?” por el número de threads utilizados, que debe coincidir con el número de cores físicos y cores lógicos utilizados.

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

11. Rellenar una tabla como la 17Tabla 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
#Ordenes para el Gestor de carga de trabajo:
#1. Asigna al trabajo un nombre
#SBATCH --job-name=bp1_10
#2. Asignar el trabajo a una partición (cola)
#SBATCH --partition=ac
#3. Asignar el trabajo a un account
#SBATCH --account=ac

#Obtener informacion 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"

```

#Instrucciones del script para ejecutar código:

```

echo -e "----SECUENCIAL----"
srun time ./SumaVectores_seq_EXE 8388608
srun time ./SumaVectores_seq_EXE 16777216
srun time ./SumaVectores_seq_EXE 33554432
srun time ./SumaVectores_seq_EXE 67108864

echo -e "----PARALLEL FOR----"
srun time ./sp-OpenMP-for_EXE 8388608
srun time ./sp-OpenMP-for_EXE 16777216
srun time ./sp-OpenMP-for_EXE 33554432
srun time ./sp-OpenMP-for_EXE 67108864

```

(RECUERDE ADJUNTAR LOS CÓDIGOS AL .ZIP)

CAPTURAS DE PANTALLA (ejecución en atcgrid):

```

>>ls
ejeri0 ejer11 ejer5
[ValentinoLugli c1estudiante16@atcgrid:~/bp1] 2021-04-08 jueves
>>cd ejer11/
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer11] 2021-04-08 jueves
>>ls
sp-OpenMP-for_EXE sp-OpenMP-script11.sh SumaVectores_seq_EXE
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer11] 2021-04-08 jueves
>>sbatch -pac -Aac -n1 -c12 --hint=nomultithread --exclusive sp-OpenMP-script11.sh
Submitted batch job 82633
[ValentinoLugli c1estudiante16@atcgrid:~/bp1/ejer11] 2021-04-08 jueves
>>

```

ejeri0: bash (c1estudiante16) atcgrid.ugr.es ejer10: sftp

Nodo de cómputo ATCGRID

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		
	Elapsed	CPU-user	CPU- sys	Elapsed	CPU-user	CPU- sys
8388608	0:00.49	0.46	0.02	0:00.06	0.52	0.10
16777216	0:00.93	0.83	0.10	0:00.10	1.01	0.18
33554432	0:01.87	1.72	0.15	0:00.18	1.66	0.37
67108864	0:03.52	3.25	0.27	0:00.33	3.17	0.73

El tiempo que se obtiene de CPU es mucho mayor al tiempo real cuando se utilizan múltiples hilos, esto se debe a que la CPU suma el tiempo que tomó cada hilo en hacer la tarea aunque este tiempo haya sido en paralelo realmente. El tiempo vuelve a ser menor que elapsed si se divide el tiempo de CPU entre los flujos de control, en este caso, $3.17 \text{ s} / 12 = 0.2642 \text{ s}$; que es menor que 0.33 s.

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 ¿? 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						
16777216						
33554432						
67108864						