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): Carlos Sánchez Páez
Grupo de prácticas: A2
Fecha de entrega: 21/03/2018
Fecha evaluación en clase: 03/04/2018

---

### 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`

```c
#include <stdio.h>
#include <stdlib.h>
#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
#endif

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`

```c
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

void funcA() {
    printf("En funcA: esta sección la ejecuta el thread %d\n", omp_get_thread_num());
}
void funcB() {
    printf("En funcB: esta sección la ejecuta el thread %d\n", omp_get_thread_num());
}

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

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`

```c
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

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("Single ejecutado por 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");
        }

    }

}
```

**CAPTURAS DE PANTALLA:**

```
[Carlos Sanchez Paez csp98@csp98-GL552VW:~/Escritorio/AC/Prácticas/practica1/src
] 2018-03-20 martes
$./singleModificado
Introduce valor de inicialización a:5
Single ejecutada por el thread 5
Single ejecutado por thread 4
Después de la región parallel
b[0]=5  b[1]=5  b[2]=5  b[3]=5  b[4]=5  b[5]=5  b[6]=5  b[7]=5  b[8]=5
[Carlos Sanchez Paez csp98@csp98-GL552VW:~/Escritorio/AC/Prácticas/practica1/src
] 2018-03-20 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`

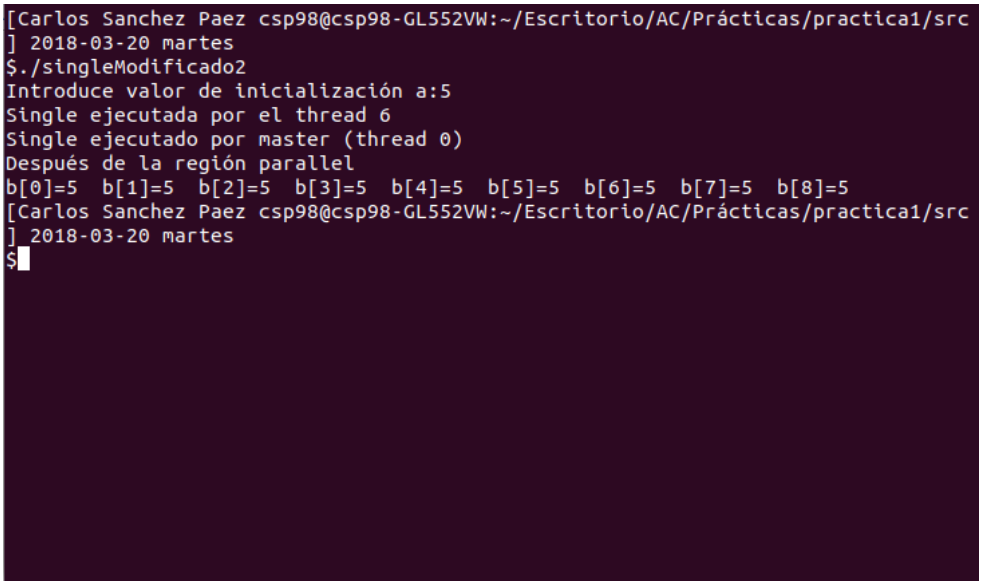
```c
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

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("Single ejecutado por master (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");
        }

    }
}
```

**CAPTURAS DE PANTALLA:**




**RESPUESTA A LA PREGUNTA:**

Que al usar una directiva *master*, el bloque de sentencias se ejecutará siempre por el thread 0 (identificador del *master*). Sin embargo, con single, cualquier thread podría ejecutarlo.

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.

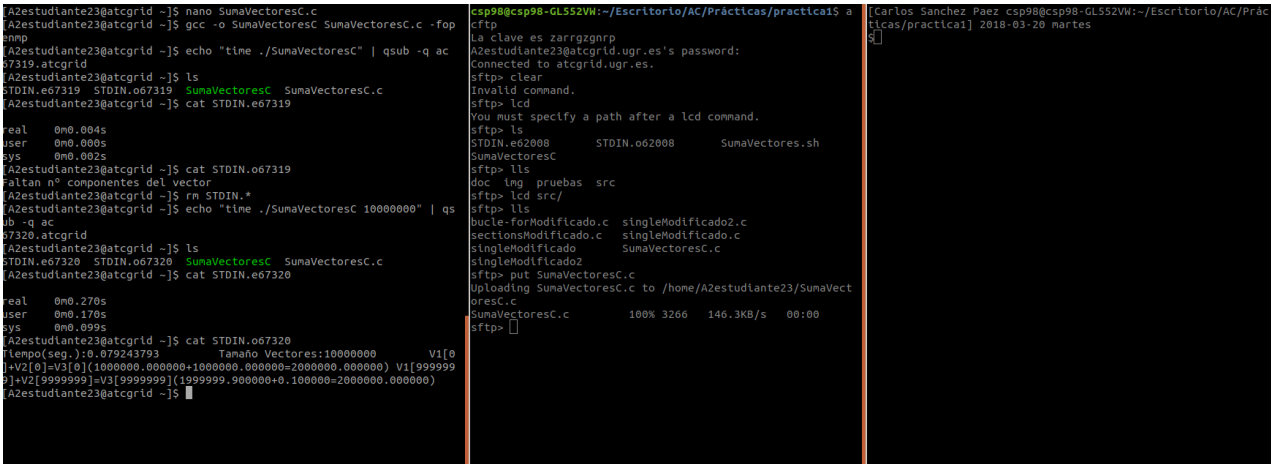
**RESPUESTA:**

Porque las instrucciones que se ejecutan después requieren que todos los threads hayan finalizado su ejecución. Por tanto, si eliminamos la barrera (punto en el que todos los threads se esperan), puede haber hebras que no hayan finalizado aún, causando un error en el programa.

## Resto de ejercicios

5. El programa secuencial C del Listado 1 calcula la suma de dos vectores (v3 = v1 + v2; v3(i) = v1(i) + v2(i), i=0,…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:**

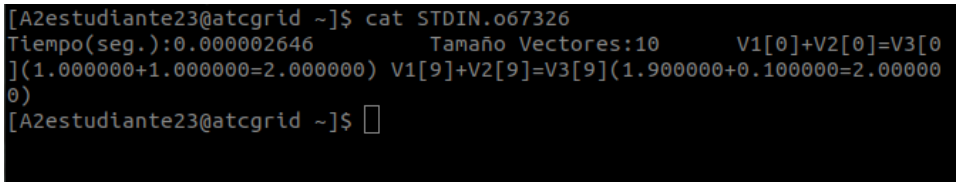


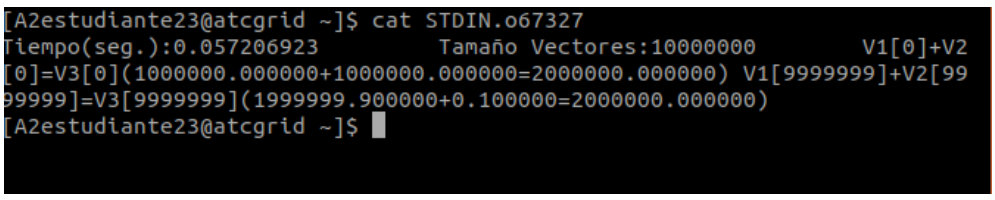

0,17+0,099=0,269 < 0,270

Es menor, porque en el tiempo real se añade el tiempo de planificación del sistema, entrada-salida, etc.

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). Incorpore **el código ensamblador de la parte de la suma de vectores** en el cuaderno.

**CAPTURAS DE PANTALLA:**



**RESPUESTA:** cálculo de los MIPS y los MFLOPS

$MIPS_{t=10}=(NI_{bucle}*N_{iteraciones}+3)/(T_{CPU}*10^6)=(6*10+3)/2{,}646=23{,}80952381$ MIPS

$MIPS_{t=10000000}=(NI_{bucle}*N_{iteraciones}+3)/(T_{CPU}*10^6)=(6*10000000+3)/57206{,}923=1048{,}824032333$ MIPS

$MFLOPS_{t=10}=Operaciones_{float\ bucle}*N_{iteraciones}/(T_{CPU}*10^6)=3*10/\ 2{,}646=11{,}337868481$ MFLOPS

$MFLOPS_{t=10000000}=Operaciones_{floatbucle}*N_{iteraciones}/(T_{CPU}*10^6)=3*10000000/57206{,}923=$

$=524{,}412054114$ MFLOPS

**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
.L5:
        movsd   v1(,%rax,8), %xmm0
        addsd   v2(,%rax,8), %xmm0
        movsd   %xmm0, v3(,%rax,8)
        addq    $1, %rax
        cmpl    %eax, %ebx
        ja      .L5
.L6:
        leaq    16(%rsp), %rsi
        xorl    %edi, %edi
        call    clock_gettime
```

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,…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

```
63      //Inicializar vectores
64      #pragma omp parallel for
65          for (i = 0; i < N; i++)
66              v1[i] = N * 0.1 + i * 0.1; v2[i] = N * 0.1 - i * 0.1; //los valores dependen de N
67      cgt1=omp_get_wtime();
68      //Calcular suma de vectores
69      #pragma omp parallel for
70          for (i = 0; i < N; i++)
71              v3[i] = v1[i] + v2[i];
72
73      cgt2=omp_get_wtime();
74      ncgt = cgt2-cgt1/ (1.e+9);
```

**(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)**

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

```
[A2estudiante23@atcgrid ~]$ cat STDIN.o67918
Tiempo(seg.):0.003878408        Tamaño Vectores:8
V1[0]+V2[0]=V3[0](0.800000+0.800000=1.600000)
V1[1]+V2[1]=V3[1](0.900000+0.000000=0.900000)
V1[2]+V2[2]=V3[2](1.000000+0.000000=1.000000)
V1[3]+V2[3]=V3[3](1.100000+0.000000=1.100000)
V1[4]+V2[4]=V3[4](1.200000+0.000000=1.200000)
V1[5]+V2[5]=V3[5](1.300000+0.000000=1.300000)
V1[6]+V2[6]=V3[6](1.400000+0.000000=1.400000)
V1[7]+V2[7]=V3[7](1.500000+0.000000=1.500000)
[A2estudiante23@atcgrid ~]$
```

```
[A2estudiante23@atcgrid ~]$ cat STDIN.o67920
Tiempo(seg.):0.004044720        Tamaño Vectores:11
V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000)
V1[1]+V2[1]=V3[1](1.200000+0.000000=1.200000)
V1[2]+V2[2]=V3[2](1.300000+0.000000=1.300000)
V1[3]+V2[3]=V3[3](1.400000+0.000000=1.400000)
V1[4]+V2[4]=V3[4](1.500000+0.000000=1.500000)
V1[5]+V2[5]=V3[5](1.600000+0.000000=1.600000)
V1[6]+V2[6]=V3[6](1.700000+0.000000=1.700000)
V1[7]+V2[7]=V3[7](1.800000+0.000000=1.800000)
V1[8]+V2[8]=V3[8](1.900000+0.000000=1.900000)
V1[9]+V2[9]=V3[9](2.000000+0.000000=2.000000)
V1[10]+V2[10]=V3[10](2.100000+0.000000=2.100000)
[A2estudiante23@atcgrid ~]$
```

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 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

```
    //Inicializar vectores
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            for (i = 0; i < N / 2; i++)
                v1[i] = N * 0.1 + i * 0.1; v2[i] = N * 0.1 - i * 0.1; //los valores dependen de N
        }
        #pragma omp section
        {
            for (i = N / 2 + 1; i < N; i++)
                v1[i] = N * 0.1 + i * 0.1; v2[i] = N * 0.1 - i * 0.1; //los valores dependen de N
        }
    }
    cgt1 = omp_get_wtime();
//Calcular suma de vectores
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            for (i = 0; i < N/2; i++)
            v3[i] = v1[i] + v2[i];
        }
        #pragma omp section
        {
            for (i = N/2+1; i < N; i++)
            v3[i] = v1[i] + v2[i];
        }
    }
    cgt2 = omp_get_wtime();
    ncgt = cgt2 - cgt1;
```

**(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)**

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

```
[A2estudiante23@atcgrid ~]$ cat STDIN.o67943
Tiempo(seg.):0.003833455        Tamaño Vectores:8
V1[0]+V2[0]=V3[0](0.800000+0.000000=0.800000)
V1[1]+V2[1]=V3[1](0.900000+0.000000=0.900000)
V1[2]+V2[2]=V3[2](1.000000+0.000000=1.000000)
V1[3]+V2[3]=V3[3](1.100000+0.000000=1.100000)
V1[4]+V2[4]=V3[4](0.000000+0.400000=0.000000)
V1[5]+V2[5]=V3[5](1.300000+0.000000=1.300000)
V1[6]+V2[6]=V3[6](1.400000+0.000000=1.400000)
V1[7]+V2[7]=V3[7](1.500000+0.000000=1.500000)
[A2estudiante23@atcgrid ~]$
```

```
[A2estudiante23@atcgrid ~]$ cat STDIN.o67944
Tiempo(seg.):0.004944089        Tamaño Vectores:11
V1[0]+V2[0]=V3[0](1.100000+0.000000=1.100000)
V1[1]+V2[1]=V3[1](1.200000+0.000000=1.200000)
V1[2]+V2[2]=V3[2](1.300000+0.000000=1.300000)
V1[3]+V2[3]=V3[3](1.400000+0.000000=1.400000)
V1[4]+V2[4]=V3[4](1.500000+0.000000=1.500000)
V1[5]+V2[5]=V3[5](0.000000+0.600000=0.000000)
V1[6]+V2[6]=V3[6](1.700000+0.000000=1.700000)
V1[7]+V2[7]=V3[7](1.800000+0.000000=1.800000)
V1[8]+V2[8]=V3[8](1.900000+0.000000=1.900000)
V1[9]+V2[9]=V3[9](2.000000+0.000000=2.000000)
V1[10]+V2[10]=V3[10](2.100000+0.000000=2.100000)
[A2estudiante23@atcgrid ~]$
```

9. ¿Cuántos threads y cuántos cores como máximo podría utilizar la versión que ha implementado en el ejercicio 7? Razone su respuesta. ¿Cuántos threads y cuantos cores como máximo podría utilizar la versión que ha implementado en el ejercicio 8? Razone su respuesta.

   **RESPUESTA:**

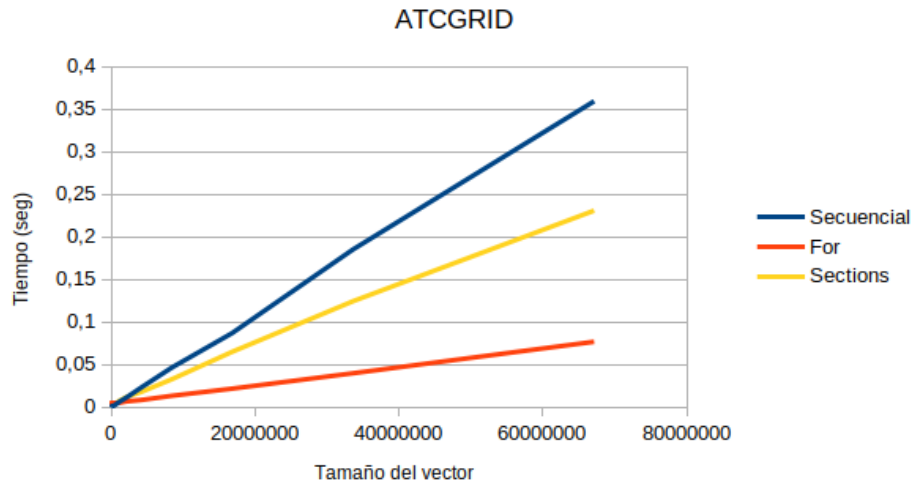
   En el ejercicio 7 (*parallel for*) se utilizarán tantos threads como permita la variable OMP_NUM_THREADS y la propia arquitectura. Sin embargo, en el ejercicio 8 también influye en número de *sections* que haya diseñado el programador, ya que aunque OMP_NUM_THREADS tome el valor máximo, se limitará el número de threads al número de *sections* programadas.

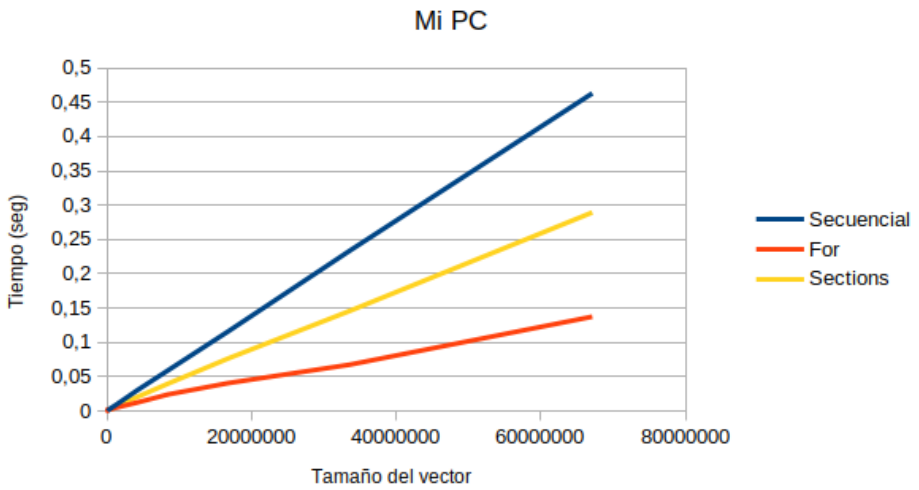
10. Rellenar una tabla como la Tabla 2 para atcgrid y otra para su PC con los tiempos de ejecución de los programas paralelos implementados en los ejercicios 7 y 8 y el programa secuencial del Listado 1. Generar los ejecutables usando -O2. En la tabla debe aparecer el tiempo de ejecución del trozo de código que realiza la suma en paralelo (este es el tiempo que deben imprimir los programas). Ponga en la tabla el número de threads/cores que usan los códigos. Represente en una gráfica los tres tiempos. NOTA: Nunca ejecute código que imprima todos los componentes del resultado cuando este número sea elevado.

    **RESPUESTA:**

| ATCGRID | | | |
|---|---|---|---|
| **Nº de Componentes** | **T. secuencial vect. Globales** **1 thread/core** | **T. paralelo (versión `for`)** **6 threads/cores** | **T. paralelo (versión `sections`)** **6 threads/cores** |
| **16384** | 0,000091567 | 0,007406861 | 0,003816963 |
| **32768** | 0,000138170 | 0,004557163 | 0,004411745 |
| **65536** | 0,000345089 | 0,003338499 | 0,004539793 |
| **131072** | 0,000688219 | 0,004049332 | 0,004449406 |
| **262144** | 0,001391591 | 0,004441273 | 0,004963458 |
| **524288** | 0,002618663 | 0,004356673 | 0,003856493 |
| **1048576** | 0,006072959 | 0,004816088 | 0,007141266 |
| **2097152** | 0,011361872 | 0,006719527 | 0,011945522 |
| **4194304** | 0,023312658 | 0,008434360 | 0,018012465 |
| **8388608** | 0,046395079 | 0,013351370 | 0,032682086 |
| **16777216** | 0,086894083 | 0,021717156 | 0,064976367 |
| **33554432** | 0,185008080 | 0,039741391 | 0,124307696 |
| **67108864** | 0,359282655 | 0,076795294 | 0,230819112 |

| MI PC | | | |
|---|---|---|---|
| **Nº de Componentes** | **T. secuencial vect. Globales** **1 thread/core** | **T. paralelo (versión `for`)** **4 threads/cores** | **T. paralelo (versión `sections`)** **4 threads/cores** |
| **16384** | 0.000114191 | 0.001549960 | 0.001439787 |
| **32768** | 0.000219413 | 0.000114191 | 0.001453126 |
| **65536** | 0.000453754 | 0.001771744 | 0.001620039 |
| **131072** | 0.000832428 | 0.001917076 | 0.002174969 |
| **262144** | 0.001770870 | 0.002248953 | 0.001574927 |
| **524288** | 0.004150900 | 0.002455461 | 0.002848738 |
| **1048576** | 0.007486943 | 0.004205928 | 0.005646633 |
| **2097152** | 0.015074341 | 0.006923681 | 0.011006473 |
| **4194304** | 0.030300146 | 0.012287634 | 0.020414176 |
| **8388608** | 0.058817209 | 0.023882004 | 0.039512883 |
| **16777216** | 0.116156945 | 0.040534236 | 0.076522421 |
| **33554432** | 0.233531140 | 0.067403596 | 0.145778975 |
| **67108864** | 0.463083358 | 0.137403739 | 0.289449440 |



11. Rellenar una tabla como la Tabla 3 para atcgrid con el tiempo de ejecución, tiempo de CPU del usuario y tiempo CPU del sistema obtenidos con `time` para el ejecutable del ejercicio 7 y para el programa secuencial del Listado 1. Ponga en la tabla el número de threads/cores que usan los códigos. ¿El tiempo de CPU que se obtiene es mayor o igual que el tiempo real (*elapsed*)? Justifique la respuesta.
**RESPUESTA**:

**MI PC**

| **Nº de Componentes** | **Tiempo secuencial vect. Globales 1 thread/core** | | | **Tiempo paralelo/versión `for` 6 Threads/cores** | | |
|---|---|---|---|---|---|---|
| | ***Elapsed*** | ***CPU-user*** | ***CPU- sys*** | ***Elapsed*** | ***CPU-user*** | ***CPU- sys*** |
| **65536** | 0m0,004s | 0m0,004s | 0m0,000s | 0m0,003s | 0m0,013s | 0m0,000s |
| **131072** | 0m0,006s | 0m0,006s | 0m0,000s | 0m0,002s | 0m0,011s | 0m0,000s |
| **262144** | 0m0,012s | 0m0,012s | 0m0,000s | 0m0,004s | 0m0,005s | 0m0,013s |
| **524288** | 0m0,025s | 0m0,004s | 0m0,020s | 0m0,004s | 0m0,000s | 0m0,021s |
| **1048576** | 0m0,047s | 0m0,047s | 0m0,000s | 0m0,006s | 0m0,028s | 0m0,004s |
| **2097152** | 0m0,093s | 0m0,080s | 0m0,012s | 0m0,012s | 0m0,024s | 0m0,040s |
| **4194304** | 0m0,186s | 0m0,142s | 0m0,044s | 0m0,021s | 0m0,053s | 0m0,062s |
| **8388608** | 0m0,368s | 0m0,244s | 0m0,124s | 0m0,043s | 0m0,088s | 0m0,132s |
| **16777216** | 0m0,766s | 0m0,591s | 0m0,176s | 0m0,082s | 0m0,172s | 0m0,240s |
| **33554432** | 0m1,468s | 0m1,096s | 0m0,372s | 0m0,142s | 0m0,232s | 0m0,610s |
| **67108864** | 0m3,059s | 0m2,427s | 0m0,632s | 0m0,276s | 0m0,516s | 0m1,175s |

**ATCGRID**

| **Nº de Componentes** | **Tiempo secuencial vect. Globales 1 thread/core** | | | **Tiempo paralelo/versión `for` 6 Threads/cores** | | |
|---|---|---|---|---|---|---|
| | ***Elapsed*** | ***CPU-user*** | ***CPU- sys*** | ***Elapsed*** | ***CPU-user*** | ***CPU- sys*** |
| **65536** | 0m0.002s | 0m0.002s | 0m0.000s | 0m0.010s | 0m0.148s | 0m0.022s |
| **131072** | 0m0.003s | 0m0.000s | 0m0.003s | 0m0.012s | 0m0.185s | 0m0.008s |
| **262144** | 0m0.004s | 0m0.002s | 0m0.003s | 0m0.011s | 0m0.166s | 0m0.021s |
| **524288** | 0m0.010s | 0m0.004s | 0m0.007s | 0m0.011s | 0m0.180s | 0m0.003s |
| **1048576** | 0m0.019s | 0m0.006s | 0m0.013s | 0m0.013s | 0m0.170s | 0m0.039s |
| **2097152** | 0m0.038s | 0m0.016s | 0m0.021s | 0m0.017s | 0m0.198s | 0m0.040s |
| **4194304** | 0m0.073s | 0m0.031s | 0m0.042s | 0m0.024s | 0m0.206s | 0m0.099s |
| **8388608** | 0m0.137s | 0m0.045s | 0m0.090s | 0m0.037s | 0m0.243s | 0m0.219s |
| **16777216** | 0m0.273s | 0m0.100s | 0m0.172s | 0m0.061s | 0m0.341s | 0m0.418s |
| **33554432** | 0m0.532s | 0m0.196s | 0m0.329s | 0m0.107s | 0m0.499s | 0m0.856s |
| **67108864** | 0m1.062s | 0m0.356s | 0m0.695s | 0m0.181s | 0m0.813s | 0m1.605s |

Cuando ejecutamos un programa secuencial (un sólo core), se cumple que $t_{elapsed}=t_{user}+t_{sys}$. Sin embargo, cuando ejecutamos código concurrente, $t_{elapsed} <= t_{user}+t_{sys}$, ya que el tiempo real mide el tiempo de una sóla hebra, mientras que el tiempo de usuario mide el tiempo que tardan todos los threads.