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): Francisco Navarro Morales
Grupo de prácticas:C2
Fecha de entrega:
Fecha evaluación en clase:

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: código fuente `bucle-forModificado.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: código fuente `sectionsModificado.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()a{
    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() ;
         }
    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: código fuente `singleModificado.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 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("Después de la región parallel:\n");
            for(i=0;i<n;i++) printf("b[%d] = %d\t",i,b[i]);
            printf("\n") ;
```

```
            printf("Single ejecutada por el thread %d\n",
                   omp_get_thread_num()) ;
        }
    }


}
```

CAPTURAS DE PANTALLA:

(Hago varias

```
pinguino 1N0 ~ UGR … Practicas 1 work % ./single
Introduce valor inicialización a:1
Single ejecutada por el thread 2
Después de la región parallel:
b[0] = 1        b[1] = 1        b[2] = 1        b[3] = 1        b[4] = 1        b[5] = 1 b[6] =
 1        b[7] = 1        b[8] = 1
Single ejecutada por el thread 1
pinguino 1N0 ~ UGR … Practicas 1 work % ./single
Introduce valor inicialización a:2
Single ejecutada por el thread 2
Después de la región parallel:
b[0] = 2        b[1] = 2        b[2] = 2        b[3] = 2        b[4] = 2        b[5] = 2 b[6] =
 2        b[7] = 2        b[8] = 2
Single ejecutada por el thread 0
pinguino 1N0 ~ UGR … Practicas 1 work % ./single
Introduce valor inicialización a:3
Single ejecutada por el thread 2
Después de la región parallel:
b[0] = 3        b[1] = 3        b[2] = 3        b[3] = 3        b[4] = 3        b[5] = 3 b[6] =
 3        b[7] = 3        b[8] = 3
Single ejecutada por el thread 3
```

ejecuciones porque es interesante ver que en cada una el primer y el segundo single se ejecutan por un thread que no tiene por qué ser el mismo).

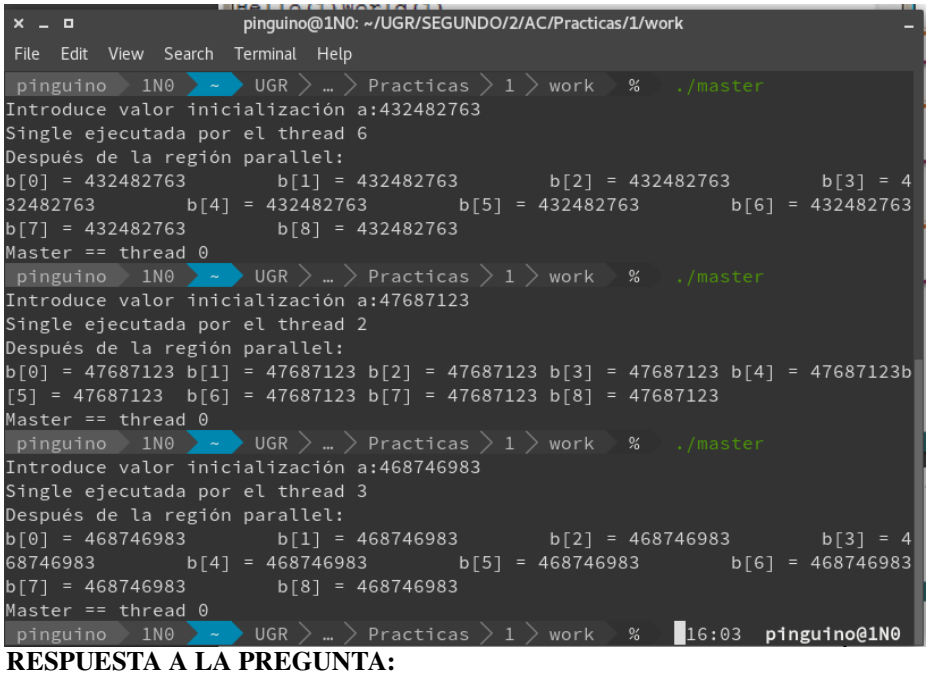
3. Imprimir los resultados del programa `single.c` usando una directiva `master` dentro de la construcción `parallel` en lugar de imprimirlos fuera de la región `parallel`. Añadir lo necesario, dentro de la nueva directiva `master` incorporada, para que se imprima el identificador del thread que ejecuta 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: código fuente `singleModificado2.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 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("Después de la región parallel:\n");
            for(i=0;i<n;i++) printf("b[%d] = %d\t",i,b[i]);
            printf("\n") ;
            printf("Master == thread %d\n",
                   omp_get_thread_num()) ;
        }
    }


}
```

CAPTURAS DE PANTALLA:



RESPUESTA A LA PREGUNTA:

La principal diferencia con respecto al código del ejercicio anterior es que, en este caso, las sentencias de la parte “master” son ejecutadas siempre por la hebra 0. Además, aunque esto no se aprecia en los volcados de pantalla, en el caso de la directiva single existe una barrera implícita al final, que no existe en la directiva master.

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:

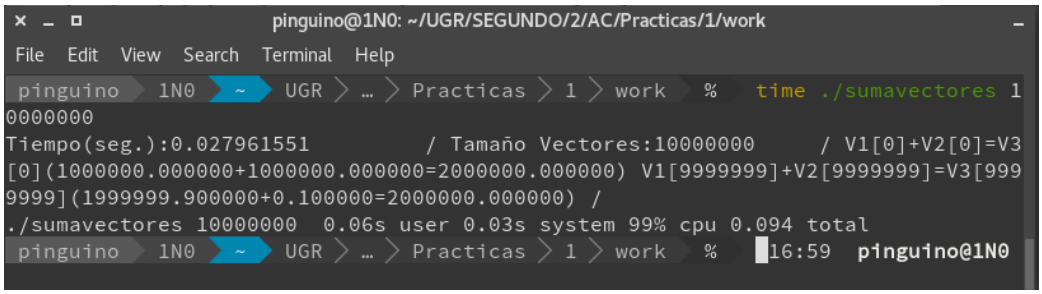
Porque la directiva barrier se encarga de sincronizar las hebras y de evitar que la hebra master imprima el resultado antes de que todas las hebras hayan sumado su sumalocal. Al eliminarla, se dan condiciones de carrera; aunque **existe una barrera implícita al final de la directiva for**, no es así con la directiva atomic y, en cuanto la hebra master sume su sumalocal, imprimirá le resultado independientemente de que las demás hebras hayan sumado o no su sumalocal.

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 el PC local, 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 mayor o igual que el tiempo real (*elapsed*)? Justifique la respuesta.

La suma del tiempo de usuario y del tiempo de cpu ser'ia 0.9s, que es menor que el tiempo total empleado (0.094). Esto se debe a algunos factores que retrasan la ejecución del programa como posibles cambios de contexto debidos a la ejecución de otros programas o esperas debidas a entrada/salida. Debido al número de procesadores de los ordenadores actuales y dado que la carga de mi sistema no es muy elevada mientras realizo las practicas, es normal que se aproveche práctamente todo el tiempo y que la diferencia sea de tan solo 0.004 segundos.

CAPTURAS DE PANTALLA:



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:**

```
        call    clock_gettime
        movl    $0, -4(%rbp)
        jmp     .L10
.L11:
        movl    -4(%rbp), %eax
        cltq
        movsd   v1(,%rax,8), %xmm1
        movl    -4(%rbp), %eax
        cltq
        movsd   v2(,%rax,8), %xmm0
        addsd   %xmm1, %xmm0
        movl    -4(%rbp), %eax
        cltq
        movsd   %xmm0, v3(,%rax,8)
        addl    $1, -4(%rbp)
.L10:
        movl    -4(%rbp), %eax
        cmpl    -8(%rbp), %eax
        jb      .L11
        leaq    -48(%rbp), %rax
        movq    %rax, %rsi
        movl    $0, %edi
        call    clock_gettime
```

En este fragmento de código encontramos:

1 instrucción movl + 1 instrucción jmp +

{por cada componente del vector –

7 mv + 1 cmp + 1 jb + 3 clqt + 2 add

} + 1 lea + 2 mov

es decir:

NI = 5+14n

dónde n es el número de componentes del vector.

```
C2estudiante12@atcgrid:~
File  Edit  View  Search  Terminal  Help
sumavectores
[C2estudiante12@atcgrid ~]$ echo "./sumavectores 10" | qsub -q ac
50411.atcgrid
[C2estudiante12@atcgrid ~]$ ls
STDIN.e50411  STDIN.o50411  sumavectores
[C2estudiante12@atcgrid ~]$ cat STDIN.o50411
Tiempo(seg.):0.000002655         / Tamaño Vectores:10   / V1[0]+V2[0]=V3[0](1.00
0000+1.000000=2.000000) V1[9]+V2[9]=V3[9](1.900000+0.100000=2.000000) /
[C2estudiante12@atcgrid ~]$ mv STDIN.o50411 10out
[C2estudiante12@atcgrid ~]$ ls
10out  STDIN.e50411  sumavectores
[C2estudiante12@atcgrid ~]$ rm STDIN.e50411
[C2estudiante12@atcgrid ~]$ echo "./sumavectores 10000000" | qsub -q ac
50412.atcgrid
[C2estudiante12@atcgrid ~]$ ls
10out  STDIN.e50412  STDIN.o50412  sumavectores
[C2estudiante12@atcgrid ~]$ cat STDIN.o50412
Tiempo(seg.):0.047345627         / Tamaño Vectores:10000000     / V1[0]+V2[0]=V3
[0](1000000.000000+1000000.000000=2000000.000000) V1[9999999]+V2[9999999]=V3[999
9999](1999999.900000+0.100000=2000000.000000) /
[C2estudiante12@atcgrid ~]$ mv STDIN.o50412 10000000out
[C2estudiante12@atcgrid ~]$ ls
10000000out  10out  STDIN.e50412  sumavectores
[C2estudiante12@atcgrid ~]$
```

```
sftp C2estudiante12@atcgrid.ugr.es
File  Edit  View  Search  Terminal  Help
sftp> lcd 1
sftp> lls
AC_seminario1_OpenMP2_5.pdf  BP1_Apellido1Apellido2Nombre_Y.odt  work
sftp> lcd work
sftp> lls
5_pantallazo.png              master.c              single.c
bucle-forModificado.c         master_pantallazo.png  single_pantallazo.png
bucleMod                      sections              sumavectores
clock_gettime_pantallazo.png  sections.c            SumaVectoresC.c
master                        single                SumaVectoresC.s
sftp> put sumavectores
Uploading sumavectores to /home/C2estudiante12/sumavectores
sumavectores                                  100% 8688   389.5KB/s   00:00
sftp> ls
10000000out     10out           STDIN.e50412    sumavectores
sftp> get 10000000out 10
10000000out  10out
sftp> get 10000000out
Fetching /home/C2estudiante12/10000000out to 10000000out
/home/C2estudiante12/10000000out              100%  198    13.5KB/s   00:00
sftp> get 10out
Fetching /home/C2estudiante12/10out to 10out
/home/C2estudiante12/10out                    100%  144     3.8KB/s   00:00
sftp>
```

Para calcular los MIPS y los MFLOPS necesitamos el tiempo de ejecución en atc_grid.

Para 10 componentes del vector se obtiene un tiempo de: 0.000002655 segundos. Tendríamos, además, 5+14*10 = 145 instrucciones = NI. Entonces:

$$\text{MIPS} = \frac{NI}{T_{CPU}\times 10^6} = \frac{145}{0.000002655\times 10^6} = 54.61393597\, MIPS$$

Para 10000000 componentes se obtiene un tiempo de 0.047345627 segundos.

Luego, teniendo 5+14*10000000 = 140000005 instrucciones = NI. Entonces:

$$\text{MIPS} = \frac{NI}{T_{CPU}\times 10^6} = \frac{140000005}{0.047345627\times 10^6} = 211.212853935\, MIPS$$

En el fragmento de código sólo se utiliza una operación con número en coma flotante, (la instrucción addsd dentro del bucle) luego habrá n instrucciones en coma flotante.

Para 10 elementos:

$$MFLOPS = \frac{Operaciones\, coma\, flotante}{T_{CPU}\times 10^6} = \frac{10}{0.000002655\times 10^6} = 3.7664\, MFLOPS$$

Para 10000000 elementos:

$$MFLOPS = \frac{Operaciones\, coma\, flotante}{T_{CPU}\times 10^6} = \frac{10000000}{0.047345627\times 10^6} = 2956.978\, MFLOPS$$

código ensamblador generado de la parte de la suma de vectores

```
                call        clock_gettime
                movl        $0, -4(%rbp)
                jmp         .L10
.L11:
                movl        -4(%rbp), %eax
                cltq
                movsd       v1(,%rax,8), %xmm1
                movl        -4(%rbp), %eax
                cltq
                movsd       v2(,%rax,8), %xmm0
                addsd       %xmm1, %xmm0
                movl        -4(%rbp), %eax
                cltq
                movsd       %xmm0, v3(,%rax,8)
                addl        $1, -4(%rbp)
.L10:
                movl        -4(%rbp), %eax
                cmpl        -8(%rbp), %eax
                jb          .L11
                leaq        -48(%rbp), %rax
                movq        %rax, %rsi
                movl        $0, %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).

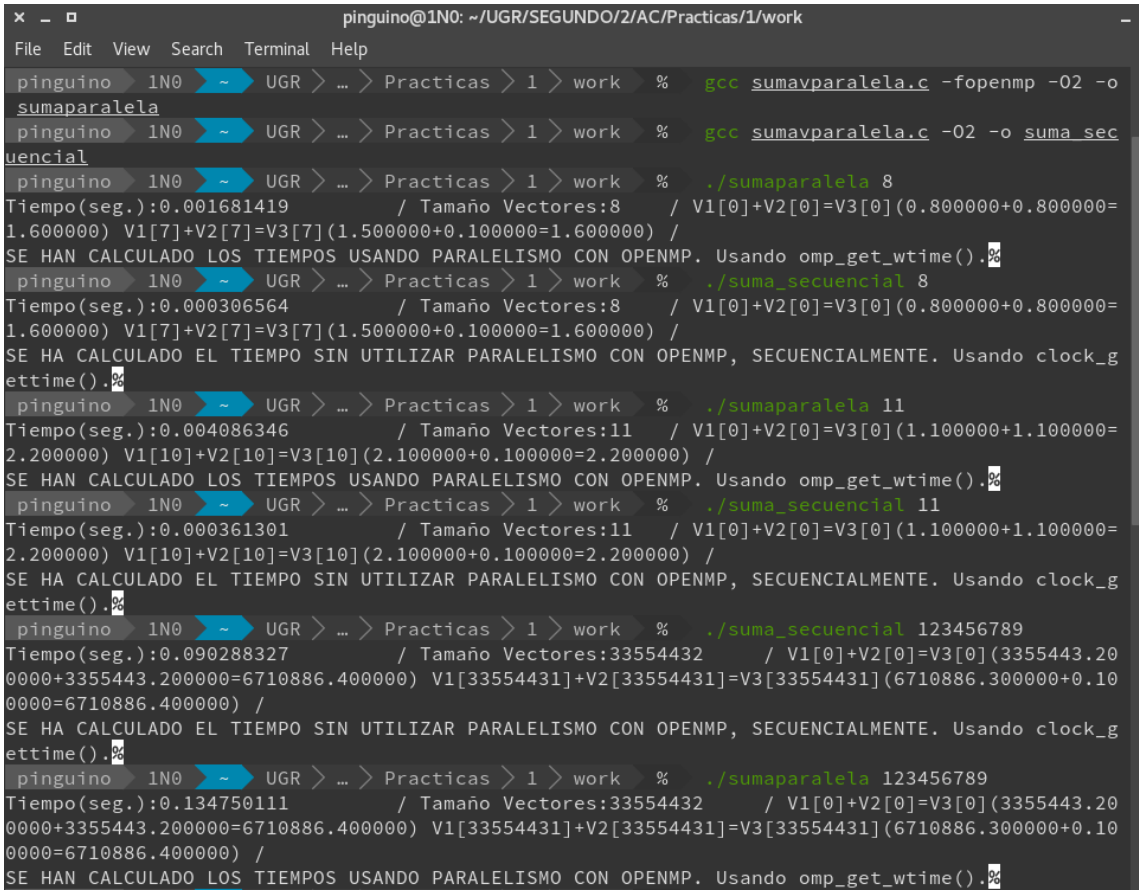
RESPUESTA: código fuente implementado

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#ifdef _OPENMP
#include<omp.h>
double vtime ;
#else
#define omp_get_thread_num() 0
struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecución
#endif


#define VECTOR_GLOBAL
#ifdef VECTOR_GLOBAL
#define MAX 33554432//=2^25
double v1[MAX], v2[MAX], v3[MAX];
#endif


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


    int i;


    //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]);
#ifdef VECTOR_LOCAL
    double v1[N], v2[N], v3[N];   // Tamaño variable local en tiempo de
ejecución ...
    // disponible en C a partir de actualización C99
#endif
#ifdef VECTOR_GLOBAL
    if (N>MAX) N=MAX;
#endif
#ifdef VECTOR_DYNAMIC
    double *v1, *v2, *v3;
    v1 = (double*) malloc(N*sizeof(double));
    v2 = (double*) malloc(N*sizeof(double));
    v3 = (double*) malloc(N*sizeof(double));
    if ( (v1==NULL) || (v2==NULL) || (v3==NULL) ){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
```

```
    }
#endif


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


#ifdef _OPENMP
    vtime = omp_get_wtime();
#else
    clock_gettime(CLOCK_REALTIME, &cgt1);
#endif
#pragma omp for
    for(i=0; i<N; i++)
        v3[i] = v1[i] + v2[i];
#ifdef _OPENMP
    vtime = omp_get_wtime() - vtime;
#else
    clock_gettime(CLOCK_REALTIME, &cgt2);
    ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
        (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
#endif


    //Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
    #ifdef _OPENMP
    printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",vtime,N);
    #else
    printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
    #endif
    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
    #ifdef _OPENMP
    printf("Tiempo(seg.):%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",
vtime,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
    printf("SE HAN CALCULADO LOS TIEMPOS USANDO PARALELISMO CON OPENMP.
Usando omp_get_wtime().") ;
    #else
    printf("Tiempo(seg.):%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]);
    printf("SE HA CALCULADO EL TIEMPO SIN UTILIZAR PARALELISMO CON OPENMP,
SECUENCIALMENTE. Usando clock_gettime().");
    #endif
#endif
#ifdef VECTOR_DYNAMIC
    free(v1); // libera el espacio reservado para v1
    free(v2); // libera el espacio reservado para v2
    free(v3); // libera el espacio reservado para v3
#endif
    return 0;
```

```
}
```

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



Como se puede apreciar en la captura, las sumas se realizan correctamente.

(AÑADIDO)

He diseñado el programa para que funcione tanto si se compila con -fopenmp como si no, y lo he compilado de las dos maneras para comprobar que, además se sumarse bien los vectores, se consiguen tiempos mejores. No obstante, para vectores pequeños (N= 8 y N=11) la diferencia entre el código secuencial y el paralelo es mínima, e incluso es peor el paralelo en ocasiones (por el tiempo que tarda en crear los threads). Por ello, para ver esto mejor, he ejecutado ambos códigos con un tamaño de N=123456789 para ver si así se notaba alguna mejoría, pero seguía sin notarse.

Tras probar distintas formas de medir los tiempos (introduciéndo las sentencias de medida detro de directivas single o midiendo todos los tiempos con clock_gettime en lugar de omp_get_wtime) sigo obteniendo los mismos resultados. Mi conclusión es que, dado que el bucle solo tiene una instrucción, por muy grande que sea el tamaño del vector, merece la pena hacerlo todo con un único thread en lugar de crear varios. Tras hacer el ejercicio 10 he comprobado que, en local, por mucho que aumente el tamaño los tiempos secuenciales parecen ser mejores (supongo que para un tamaño que tiena a infinito los tiempos paralelos superarían a los secuenciales). No obstante, en atcgrid los tiempos paralelos siempre son mejores, luego la conclusión a extraer es que mi portatil no aprovecha tan bien el paralelismo como para que sea rentable hacerlo en bucles tan simples. Quizá también pueda tener que ver el tipo de paralelismo que el procesador de mi portatil implemente.

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: código fuente implementado

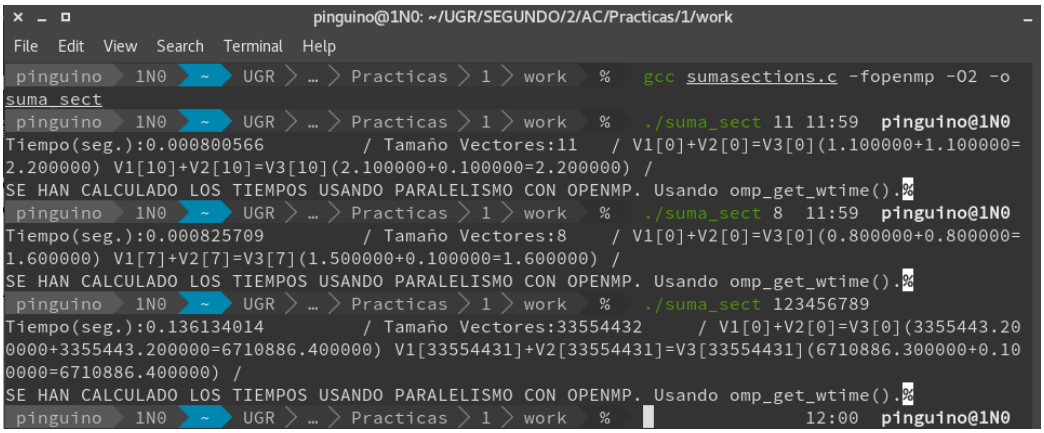
```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#ifdef _OPENMP
#include<omp.h>
double vtime ;
#else
#define omp_get_thread_num() 0
struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecución
#endif


#define VECTOR_GLOBAL
#ifdef VECTOR_GLOBAL
#define MAX 33554432//=2^25
double v1[MAX], v2[MAX], v3[MAX];
#endif


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


    int i;


    //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]);
#ifdef VECTOR_LOCAL
    double v1[N], v2[N], v3[N];   // Tamaño variable local en tiempo de
ejecución ...
    // disponible en C a partir de actualización C99
#endif
#ifdef VECTOR_GLOBAL
    if (N>MAX) N=MAX;
#endif
#ifdef VECTOR_DYNAMIC
    double *v1, *v2, *v3;
    v1 = (double*) malloc(N*sizeof(double));
    v2 = (double*) malloc(N*sizeof(double));
```

```
    v3 = (double*) malloc(N*sizeof(double));
    if ( (v1==NULL) || (v2==NULL) || (v3==NULL) ){
        printf("Error en la reserva de espacio para los vectores\n");
        exit(-2);
    }
#endif


    int Nn = N/4 ;
    // INICIALIZAR VECTORES
#pragma omp parallel
    {
#pragma omp sections
        {
#pragma omp section
            {
                for(int i = 0 ; i < Nn ; i++){
                    v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
                }
            }
#pragma omp section
            {
                for(int i = Nn ; i < 2*Nn ; i++){
                    v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
                }
            }
#pragma omp section
            {
                for(int i = 2*Nn ; i < 3*Nn ; i++){
                    v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
                }
            }
#pragma omp section
            {
                for(int i = 3*Nn ; i < N ; i++){
                    v1[i] = N*0.1+i*0.1; v2[i] = N*0.1-i*0.1;
                }
            }


        }
#pragma omp single
        {
#ifdef _OPENMP
            vtime = omp_get_wtime();
#else
            clock_gettime(CLOCK_REALTIME, &cgt1);
#endif
        }


#pragma omp sections
        {
#pragma omp section
            {
                for(int i = 0 ; i < Nn ; i++)
                    v3[i] = v1[i] + v2[i];
            }
#pragma omp section
            {
                for(int i = Nn ; i < 2*Nn ; i++)
```

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


        }
#pragma omp single
        {
#ifdef _OPENMP
            vtime = omp_get_wtime() - vtime;
#else
            clock_gettime(CLOCK_REALTIME, &cgt2);
            ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
                (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
#endif
        }
    }


    //Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
    #ifdef _OPENMP
    printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",vtime,N);
    #else
    printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",ncgt,N);
    #endif
    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
    #ifdef _OPENMP
    printf("Tiempo(seg.):%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",
vtime,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
    printf("SE HAN CALCULADO LOS TIEMPOS USANDO PARALELISMO CON OPENMP.
Usando omp_get_wtime().") ;
    #else
    printf("Tiempo(seg.):%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]);
    printf("SE HA CALCULADO EL TIEMPO SIN UTILIZAR PARALELISMO CON OPENMP,
SECUENCIALMENTE. Usando clock_gettime().");
    #endif
#endif




#ifdef VECTOR_DYNAMIC
    free(v1); // libera el espacio reservado para v1
```

```
    free(v2); // libera el espacio reservado para v2
    free(v3); // libera el espacio reservado para v3
#endif
    return 0;
}
```

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



En este caso los resultados de la suma también son correctos, y también se obtienen tiempos del orden de 0.1 segundo para tamaño N=123456789. Supongo que por las mismas razones que en el caso anterior.

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 la versión del ejercicio 7 podrían utilizarse todos los cores y threads de que se dispusiera porque el propio OpenMP se encargaría de distribuirles el trabajo. Sin embargo, en la versión del ejercicio 8 he creado solo cuatro secciones a ejecutar en paralelo y más de cuatro threads serían absurdos (y contraproducentes) porque se asignaría una sección a cada uno de los cuatro primeros threads que fueran tomados y el resto quedaría ocioso.

10. Rellenar una tabla para atcgrid y otra para el PC local 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 en atcgrid código que imprima todos los componentes del resultado.

    RESPUESTA:

Tabla 2. Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas en LOCAL.

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	0.000188	0.005576	0.005132
32768	0.000210	0.002012	0.004438
65536	0.000312	0.006869	0.000838
131072	0.000502	0.002400	0.004012
262144	0.001095	0.003088	0.001948
524288	0.001930	0.007902	0.003119
1048576	0.003559	0.011260	0.004616
2097152	0.006665	0.013258	0.012455
4194304	0.017623	0.023313	0.022239
8388608	0.025550	0.034215	0.033865
16777216	0.050730	0.067791	0.067106
33554432	0.103983	0.137866	0.133502
67108864	0.202278	0.270689	0.263114

Illustration 1: Tiempos en local

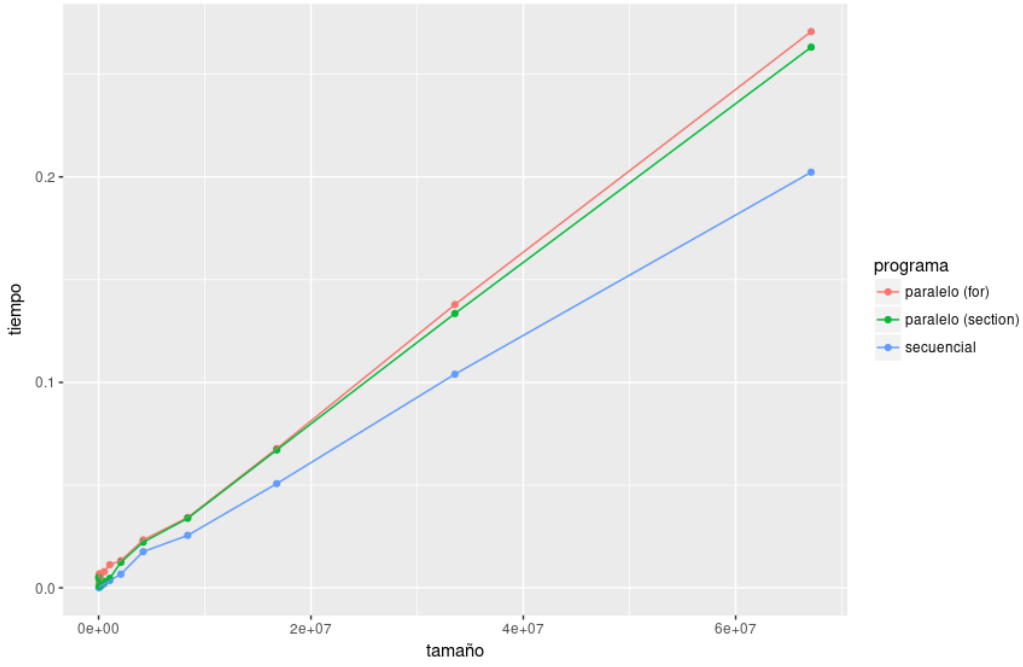
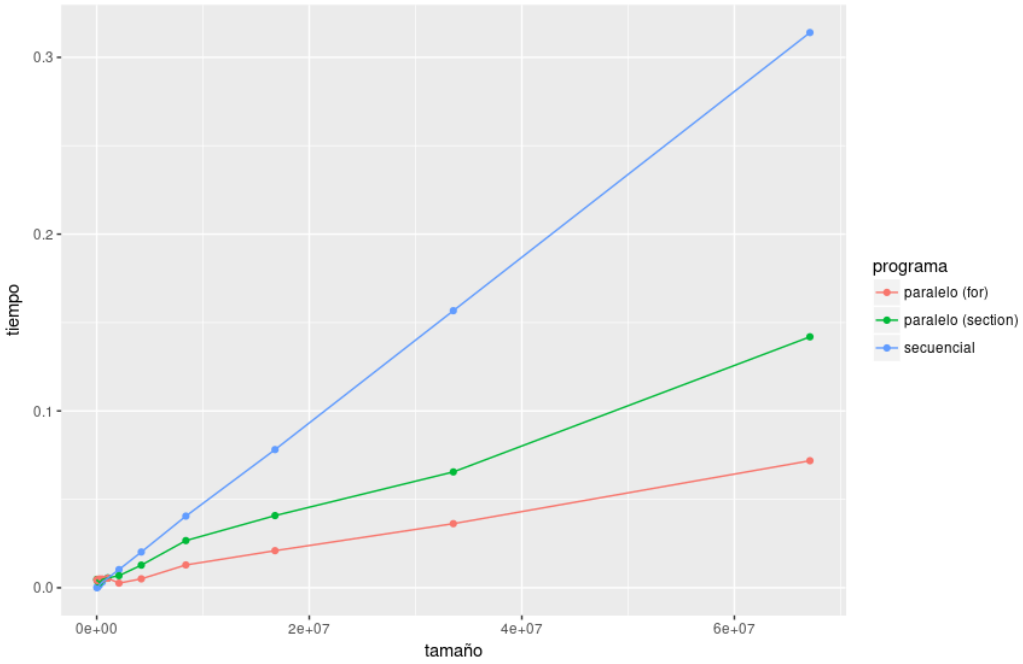


Tabla 2. Tiempos de ejecución de la versión secuencial de la suma de vectores y de las dos versiones paralelas. Sustituir en el encabezado de la tabla "¿?" por el número de threads utilizados, que debe coincidir con el número de cores físicos utilizados.

Nº de Componentes	T. secuencial vect. Globales 1 thread/core	T. paralelo (versión `for`) ¿?threads/cores	T. paralelo (versión `sections`) ¿?threads/cores
16384	0.000095	0.004384	0.004777
32768	0.000191	0.004465	0.004517
65536	0.000376	0.004230	0.003706
131072	0.000758	0.004295	0.003894
262144	0.001540	0.004964	0.002887
524288	0.003194	0.004941	0.004689
1048576	0.005525	0.005382	0.005556
2097152	0.010287	0.002610	0.006851
4194304	0.020224	0.002610	0.012802
8388608	0.040532	0.012911	0.026684
16777216	0.078145	0.020974	0.040822
33554432	0.156709	0.036260	0.065524
67108864	0.314009	0.071851	0.141927

Illustration 2: tiempos en atcgrid



11. Rellenar una tabla como la Tabla 3 para el PC local 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:

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/core**			**Tiempo paralelo/versión `for` 4 Threads/cores**		
	Elapsed	***CPU-user***	***CPU- sys***	***Elapsed***	***CPU-user***	***CPU- sys***
65536	0m0.001s	0m0.000s	0m0.000s	0m0.015s	0m0.047s	0m0.037s
131072	0m0.001s	0m0.000s	0m0.000s	0m0.007s	0m0.037s	0m0.010s
262144	0m0.003s	0m0.000s	0m0.000s	0m0.008s	0m0.050s	0m0.007s
524288	0m0.006s	0m0.000s	0m0.003s	0m0.016s	0m0.063s	0m0.033s
1048576	0m0.009s	0m0.007s	0m0.000s	0m0.027s	0m0.130s	0m0.030s
2097152	0m0.019s	0m0.017s	0m0.000s	0m0.036s	0m0.160s	0m0.040s
4194304	0m0.034s	0m0.017s	0m0.017s	0m0.055s	0m0.240s	0m0.083s
8388608	0m0.062s	0m0.040s	0m0.020s	0m0.091s	0m0.510s	0m0.150s
16777216	0m0.120s	0m0.087s	0m0.033s	0m0.175s	0m0.947s	0m0.323s
33554432	0m0.235s	0m0.160s	0m0.073s	0m0.329s	0m1.863s	0m0.603s
67108864	0m0.464s	0m0.313s	0m0.150s	0m0.640s	0m3.697s	0m1.187s

El tiempo elapsed siempre es mayor que el de CPU, ya que contiene el tiempo de CPU asociado al usuario, el de CPU asociado al sistema y un último tiempo (no reflejado en la tabla y que no viene dado directamente por la orden time) que es debido a las entradas/salidas y a la ejecución de otros programas.