

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): Daniel Bolaños Martínez

Grupo de prácticas: A1

Fecha de entrega: 30/03/2017

Fecha evaluación en clase: 31/03/2017

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>
#include <omp.h>

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\n", omp_get_thread_num(), i);

    return(0);
}
```

RESPUESTA: código fuente `sectionsModificado.c`

```
#include <stdio.h>
#include <omp.h>

void funcA() {
    printf("En funcA: esta sección la ejecuta el thread\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: código fuente `singleModificado.c`

```

#include <stdio.h>
#include <omp.h>

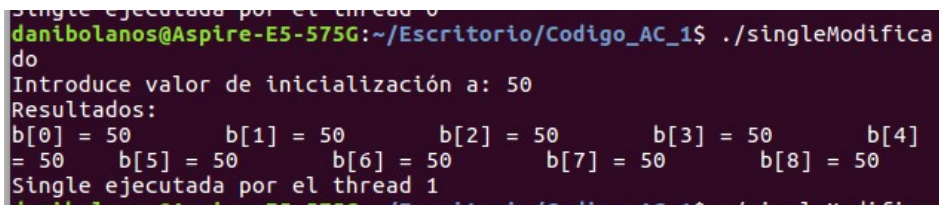
int main() {
    int n = 9, i, a, b[n];

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

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

        #pragma omp single
        { printf("Resultados:\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:


```

Single ejecutada por el thread 1
danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_1$ ./singleModifica
do
Introduce valor de inicialización a: 50
Resultados:
b[0] = 50      b[1] = 50      b[2] = 50      b[3] = 50      b[4]
= 50      b[5] = 50      b[6] = 50      b[7] = 50      b[8] = 50
Single ejecutada por el thread 1

```

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>
#include <omp.h>

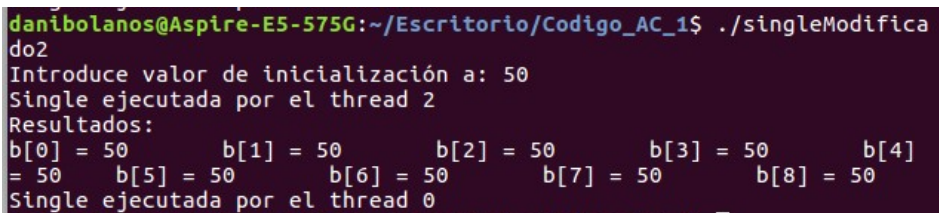
int main() {
    int n = 9, i, a, b[n];

    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("Resultados:\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:


```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_1$ ./singleModifica
do2
Introduce valor de inicialización a: 50
Single ejecutada por el thread 2
Resultados:
b[0] = 50      b[1] = 50      b[2] = 50      b[3] = 50      b[4]
= 50      b[5] = 50      b[6] = 50      b[7] = 50      b[8] = 50
Single ejecutada por el thread 0

```

RESPUESTA A LA PREGUNTA:

Como utilizamos la directiva master, los resultados del programa los ejecuta la thread 0.

4. ¿Por qué si se elimina directiva barrier en el ejemplo master.c la suma que se calcula e imprime no siempre es correcta? Responda razonadamente.

RESPUESTA:

Porque al eliminar la directiva barrier, se elimina la barrera implícita que espera a que terminen de completarse las sumas locales de algunas hebras y por tanto, puede imprimir resultados erróneos.

Resto de ejercicios

5. El programa secuencial C del Listado 1 calcula la suma de dos vectores ($v3 = v1 + v2$; $v3(i) = v1(i) + v2(i)$, $i=0, \dots, N-1$). 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 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.

CAPTURAS DE PANTALLA:

Compilado con: `$gcc -O2 SumaVectoresC.c -o SumaVectoresC -lrt`



```

danibolanos@Aspire-E5-575G:~/Escritorio/Codigo_AC_1$ time ./SumaVectoresC 10000000
00
Tiempo(seg.):0.027509279 / 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) /
real    0m0.086s
user    0m0.064s
sys     0m0.020s

```

Como podemos observar, la suma de los tiempos de usuario y sistema es menor que el tiempo real. Esto se debe a que esos 0.002s de más en el tiempo real son causados por la espera debidas a operaciones de I/O o ejecución de otros programas.

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.

RESPUESTA:

Primeramente, he generado con el código SumaVectoresC dos programas, el primero compilando con optimización -O2 y el segundo añadiendo, tal y como ha propuesto la profesora en clase, la directiva #omp pragma simd y compilando con -fopenmp.

He obtenido el código en ensamblador y el ejecutable correspondientes en cada caso y he procedido a la ejecución de ambos programas en el atcgrid para obtener su tiempo de CPU.

CAPTURAS DE PANTALLA:

```
dantbolanos@Aspire-E5-575G:~$ ssh E1estudiante20@atcgrid.ugr.es
E1estudiante20@atcgrid.ugr.es's password:
Last login: Sat Mar 25 18:29:57 2017 from 172.20.241.155
[E1estudiante20@atcgrid ~]$ ls
BP0 BP1
[E1estudiante20@atcgrid ~]$ cd BP1
[E1estudiante20@atcgrid BP1]$ ls
ejecuciones SumaVectoresC SumaVectoresOMP
[E1estudiante20@atcgrid BP1]$ echo 'BP1/SumaVectoresC 10' | qsub -q ac
52767.atcgrid
[E1estudiante20@atcgrid BP1]$ echo 'BP1/SumaVectoresC 10000000' | qsub
-q ac
52768.atcgrid
[E1estudiante20@atcgrid BP1]$ cat STDIN.o52767
Tiempo(seg.):0.000003393 / Tamaño Vectores:10 / V1[0]+V2[0]=V
3[0](1.000000+1.000000=2.000000) / / V1[9]+V2[9]=V3[9](1.900000+0.10000
0=2.000000) /
[E1estudiante20@atcgrid BP1]$ cat STDIN.o52768
Tiempo(seg.):0.047436456 / Tamaño Vectores:10000000 / V1[0]
+V2[0]=V3[0](1000000.000000+1000000.000000=2000000.000000) / / V1[99999
99]+V2[9999999]=V3[9999999](1999999.900000+0.100000=2000000.000000) /
[E1estudiante20@atcgrid BP1]$ echo 'BP1/SumaVectoresOMP 10' | qsub -q a
c
52769.atcgrid
[E1estudiante20@atcgrid BP1]$ echo 'BP1/SumaVectoresOMP 10000000' | qsu
b -q ac
52770.atcgrid
[E1estudiante20@atcgrid BP1]$ cat STDIN.o52769
Tiempo(seg.):0.000003330 / Tamaño Vectores:10 / V1[0]+V2[0]=V
3[0](1.000000+1.000000=2.000000) / / V1[9]+V2[9]=V3[9](1.900000+0.10000
0=2.000000) /
[E1estudiante20@atcgrid BP1]$ cat STDIN.o52770
Tiempo(seg.):0.045382439 / Tamaño Vectores:10000000 / V1[0]
+V2[0]=V3[0](1000000.000000+1000000.000000=2000000.000000) / / V1[99999
99]+V2[9999999]=V3[9999999](1999999.900000+0.100000=2000000.000000) /
```

Como podemos observar, en cuanto a tiempos de CPU se refiere, la diferencia en la ejecución de cada programa es casi inapreciable.

RESPUESTA: cálculo de los MIPS y los MFLOPS

$$\text{MIPS} = \text{NI} / T_{\text{CPU}} * 10^6$$

$$\text{MFLOPS} = \text{Operaciones_coma_Flotante} / T_{\text{CPU}} * 10^6$$

- SIN DIRECTIVAS:

$$\text{NI} = 3 + (6 * N^{\circ}_\text{Componentes})$$

$$\text{Operaciones_coma_Flotante} = 1 * N^{\circ}_\text{Componentes}$$

// SumaVectoresC con 10 componentes:

$$T_{\text{CPU}} = 0.000003393$$

$$\text{MIPS} = 3 + (6 * 10) / 0.000003393 * 10^6 = 18.568$$

$$\text{MFLOPS} = 1 \cdot 10^7 / 0.000003393 \cdot 10^6 = 2.947$$

// SumaVectoresC con 10⁷ componentes:

$$T_{\text{CPU}} = 0.047436456$$

$$\text{MIPS} = 3 + (6 \cdot 10^7) / 0.047436456 \cdot 10^6 = 1264.85$$

$$\text{MFLOPS} = 1 \cdot 10^7 / 0.047436456 \cdot 10^6 = 210.808$$

RESPUESTA: código ensamblador generado de la parte de la suma de vectores

```
# Fragmento ensamblador programa SumaVectoresC.c obtenido con
$gcc -O2 SumaVectoresC.c -S -o SumaVectoresC.s -lrt
```

```
call    clock_gettime
xorl    %eax, %eax
.p2align 4,,10
.p2align 3
.L5:
    movsd    v1(%rax), %xmm0
    addq     $8, %rax
    addsd    v2-8(%rax), %xmm0
    movsd    %xmm0, v3-8(%rax)
    cmpq     %rax, %rbx
    jne      .L5
.L6:
    leaq     16(%rsp), %rsi
    xorl     %edi, %edi
    call     clock_gettime
```

```
# Fragmento ensamblador programa SumaVectoresC.c con la directiva
#pragma omp simd y obtenido con :
$gcc -O2 -fopenmp-simd SumaVectoresC.c -S -o SumaVectoresOMP.s -lrt
```

```
call    clock_gettime
leal     -2(%rbx), %edx
shrl     %edx
addl     $1, %edx
cmpl     $1, %ebx
leal     (%rdx,%rdx), %eax
je       .L10
xorl     %ecx, %ecx
xorl     %esi, %esi
.L6:
    movapd   v1(%rcx), %xmm0
    addl     $1, %esi
    addq     $16, %rcx
    addpd    v2-16(%rcx), %xmm0
    movaps   %xmm0, v3-16(%rcx)
    cmpl     %esi, %edx
    ja       .L6
    cmpl     %ebx, %eax
    je       .L8
.L5:
    cltq
    movsd    v2(,%rax,8), %xmm0
    addsd    v1(,%rax,8), %xmm0
    movsd    %xmm0, v3(,%rax,8)
.L8:
```

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 ($v3 = v1 + v2$; $v3(i) = v1(i) + v2(i)$, $i=0, \dots, N-1$) 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: código fuente implementado

```
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
#include <omp.h>

// #define PRINTF_ALL // comentar para quitar el printf ...

// Hecho con Vectores Globales
#define MAX 33554432 // = 2^25
double v1[MAX], v2[MAX], v3[MAX];

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

    omp_set_num_threads(2);
    // omp_set_num_threads(12);

    int i;
    double cgt1, cgt2;
    double ncgt; // para tiempo de ejecución
    // 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)

    if (N > MAX) N = MAX;

    #pragma omp parallel
    {
```



```

#pragma omp for
//Inicializar vectores
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
}

cgt1 = omp_get_wtime();
#pragma omp parallel
{
    #pragma omp for
    //Calcular suma de vectores
    for(i=0; i<N; i++)
        v3[i] = v1[i] + v2[i];
}

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

#ifdef PRINTF_ALL
//Imprimir resultado de la suma y el tiempo de ejecución
printf("Componentes del vector v3: ");
for(i=0; i<N; i++)
    printf("\n%f", v3[i]);
printf("\n");
#endif

    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]);

return 0;
}

```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

```

danibolanos@Aspire-E5-575G:~/Escritorio/8P1_BolanosMartinezDaniel_1/Cod
igo_Scripts$ gcc -O2 -fopenmp SumaVectoresFOR.c -o SumaVectoresFOR -lrt
danibolanos@Aspire-E5-575G:~/Escritorio/8P1_BolanosMartinezDaniel_1/Cod
igo_Scripts$ ./SumaVectoresFOR 8
Componentes del vector v3:
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
Tiempo(seg.):0.001337151 / Tamaño Vectores:8 / V1[0]+V2[0]=V
3[0](0.800000+0.800000=1.600000) / / V1[7]+V2[7]=V3[7](1.500000+0.10000
0=1.600000) /
danibolanos@Aspire-E5-575G:~/Escritorio/8P1_BolanosMartinezDaniel_1/Cod
igo_Scripts$ ./SumaVectoresFOR 11
Componentes del vector v3:
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
Tiempo(seg.):0.001745279 / Tamaño Vectores:11 / V1[0]+V2[0]=V
3[0](1.100000+1.100000=2.200000) / / V1[10]+V2[10]=V3[10](2.100000+0.10
0000=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, $N = 8$ 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> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h> // biblioteca donde se encuentra la función printf()
#include <time.h> // biblioteca donde se encuentra la función clock_gettime()
#include <omp.h>

// #define PRINTF_ALL // comentar para quitar el printf ...

// Hecho con Vectores Globales
#define MAX 33554432 // = 2^25
double v1[MAX], v2[MAX], v3[MAX];

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

    omp_set_num_threads(2);
    // omp_set_num_threads(12);

    int i, j;
    double cgt1, cgt2;
    double ncgt; // para tiempo de ejecución
    // 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)

    if (N > MAX) N = MAX;

    #pragma omp parallel
    {
        #pragma omp sections
```

```

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

cgt1 = omp_get_wtime();

#pragma omp parallel
{
#pragma omp sections
{
    //Calcular suma de vectores
#pragma omp section
    for(i=0; i<N/2; i++)
        v3[i] = v1[i] + v2[i];
#pragma omp section
    for(j=N/2; j<N; j++)
        v3[j] = v1[j] + v2[j];
}
}

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

#ifdef PRINTF_ALL
//Imprimir resultado de la suma y el tiempo de ejecución
printf("Componentes del vector v3: ");
for(i=0; i<N; i++)
    printf("\n%f", v3[i]);
printf("\n");
#endif

    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]);

    return 0;
}

```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

```

danielbolanos@Aspire-E5-575G:~/Escritorio/BP1_BolanosMartinezDaniel_1/Cod
igo_Scripts$ gcc -O2 -fopenmp SumaVectoresSECTIONS.c -o SumaVectoresSEC
TIONS -lrt
danielbolanos@Aspire-E5-575G:~/Escritorio/BP1_BolanosMartinezDaniel_1/Cod
igo_Scripts$ ./SumaVectoresSECTIONS 8
Componentes del vector v3:
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
1.600000
Tiempo(seg.):0.000790759 / Tamaño Vectores:8 / V1[0]+V2[0]=V
3[0](0.800000+0.800000=1.600000) / / V1[7]+V2[7]=V3[7](1.500000+0.1000
0=1.600000) /

```

```

danibolanos@Aspire-E5-575G:~/Escritorio/BP1_BolanosMartinezDaniel_1/Cod
igo_Scripts$ ./SumaVectoresSECTIONS 11
Componentes del vector v3:
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
2.200000
Tiempo(seg.):0.000554424 / Tamaño Vectores:11 / V1[0]+V2[0]=V
3[0](1.100000+1.100000=2.200000) / / V1[10]+V2[10]=V3[10](2.100000+0.10
0000=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.

RESPUESTA:

En el ejercicio 7, utilizamos la directiva for, por tanto, el máximo número de hebras que podría utilizar este programa será igual al número que se especifique, por ejemplo con la orden:

```
$ export OMP_NUM_THREADS.
```

En el ejercicio 8, utilizamos la directiva sections, por tanto, el máximo número de hebras que podría utilizar este programa al mismo tiempo es el equivalente al número de section del código en alto nivel, que en mi caso es igual a 2. En total utiliza 4 hebras, 2 para inicializar, que elimina una vez acaba, y otras 2 para realizar la suma.

El número de cores lo especificará el número de cores físicos de la máquina en la que se ejecute el código, en mi caso 2.

10. Rellenar una tabla como la Tabla 2 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:

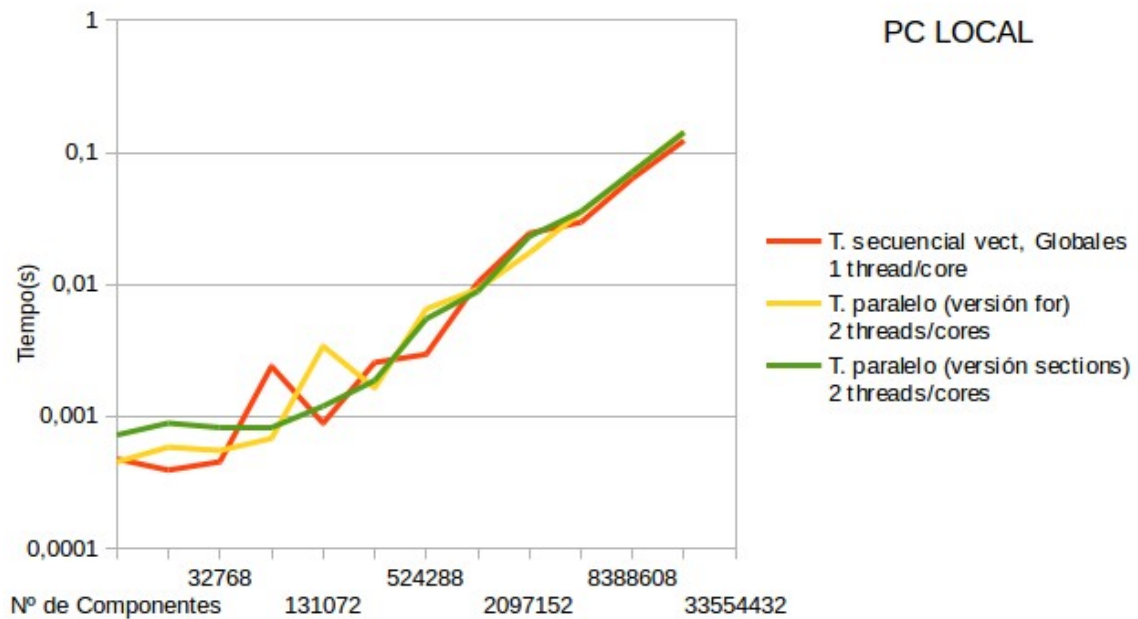
Mientras que en mi ordenador (utilizando 2 threads) las tres versiones tienen unos tiempos equivalentes, en el nodo de atcgrid (12 threads) el tiempo de ejecución usando las directivas sections y for mejora respecto a la versión sin directivas. Sobre todo en el caso del for, el cual utiliza el número de threads especificadas, al contrario que el sections que está limitado por el número de section programados.

Por ello, a modo de conclusión, podemos decir que a mayor número de threads que nos permita usar nuestra máquina, la programación paralela será mucho más eficiente que la secuencial.

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.

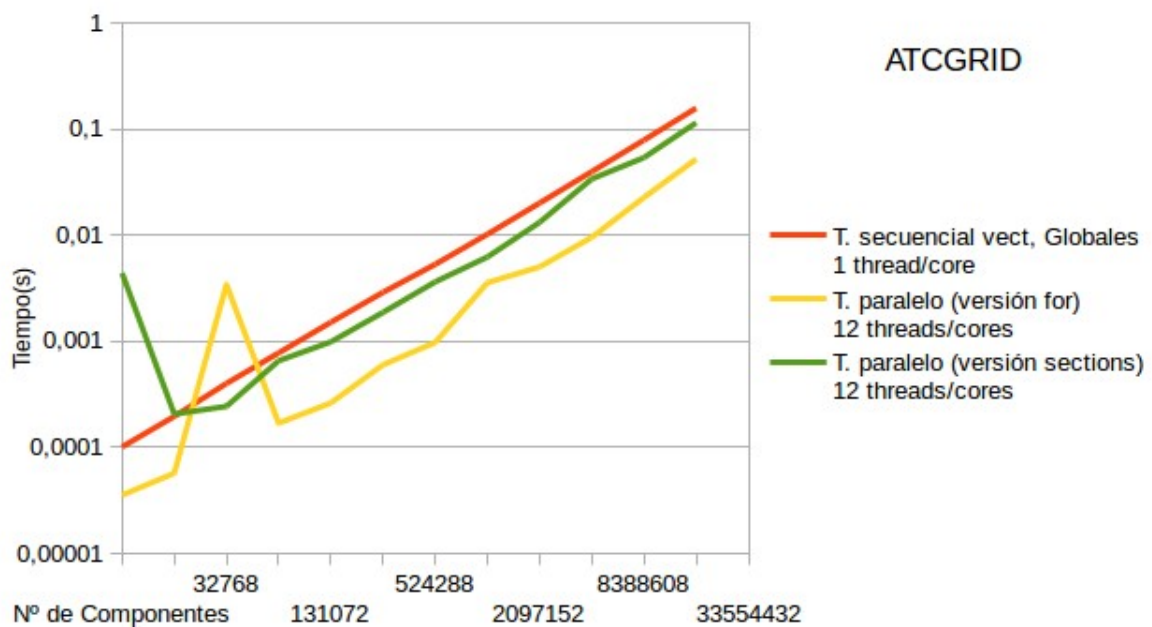
PC Local: Intel® Core™ i5-7200U CPU @ 2.50GHz

Nº de Componentes	T. secuencial vect. Globales 1 thread/core	T. paralelo (versión for) 2 threads/cores	T. paralelo (versión sections) 2 threads/cores
16384	0.000474438	0.000450336	0.000721179
32768	0.000390841	0.000583176	0.000887123
65536	0.000451389	0.000550030	0.000831858
131072	0.002388003	0.000678809	0.000820048
262144	0.000884626	0.003401776	0.001190166
524288	0.002556001	0.001638446	0.001862813
1048576	0.002945523	0.006509138	0.005472285
2097152	0.010341078	0.009193365	0.008841603
4194304	0.024380807	0.017344091	0.023086322
8388608	0.029539305	0.035068348	0.035608477
16777216	0.063572452	0.071043398	0.071101402
33554432	0.122932949	0.143585925	0.141818763
67108864	0.177509259	0.207585650	0.215188439



Atcgrid: Intel® Xeon® CPU E5645 @ 2.40GHz

Nº de Componentes	T. secuencial vect. Globales 1 thread/core	T. paralelo (versión for) 12 threads/cores	T. paralelo (versión sections) 12 threads/cores
16384	0.000101306	0.000035545	0.004423646
32768	0.000196371	0.000057286	0.000206854
65536	0.000401639	0.003454644	0.000244372
131072	0.000779847	0.000169855	0.000657789
262144	0.001517639	0.000263223	0.000992896
524288	0.002914989	0.000602277	0.001885913
1048576	0.005356128	0.000979930	0.003660377
2097152	0.010298767	0.003600482	0.006285857
4194304	0.020287683	0.005063197	0.013347793
8388608	0.040067679	0.009639883	0.034207691
16777216	0.079571526	0.022883192	0.054416554
33554432	0.159278067	0.052683778	0.115929065
67108864	0.207921727	0.116341676	0.173657769



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:

En la versión secuencial, el tiempo de *CPU* es menor que el real, esto se debe como dijimos en el ejercicio 5, a que el programa usa una sola *thread* y por tanto el tiempo real equivale al de *CPU*, al que se le restan las pérdidas por operaciones de I/O o ejecución de programas.

En la versión paralela, el tiempo de *CPU* supera al real, esto se debe a que el tiempo de *CPU* es igual a la suma de los tiempos de cada *thread* utilizada (2 en este caso), mientras que el tiempo real corresponde al tiempo de ejecución del programa.

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 2 threads/cores		
	<i>Elapsed</i>	<i>CPU-user</i>	<i>CPU- sys</i>	<i>Elapsed</i>	<i>CPU-user</i>	<i>CPU- sys</i>
65536		real 0m0.007s user 0m0.000s sys 0m0.004s			real 0m0.004s user 0m0.000s sys 0m0.000s	
131072		real 0m0.009s user 0m0.000s sys 0m0.008s			real 0m0.007s user 0m0.004s sys 0m0.000s	
262144		real 0m0.019s user 0m0.008s sys 0m0.008s			real 0m0.008s user 0m0.004s sys 0m0.004s	
524288		real 0m0.024s user 0m0.020s sys 0m0.004s			real 0m0.010s user 0m0.012s sys 0m0.000s	
1048576		real 0m0.035s user 0m0.032s sys 0m0.000s			real 0m0.013s user 0m0.008s sys 0m0.012s	
2097152		real 0m0.048s user 0m0.028s sys 0m0.016s			real 0m0.022s user 0m0.028s sys 0m0.008s	
4194304		real 0m0.079s user 0m0.056s sys 0m0.020s			real 0m0.041s user 0m0.048s sys 0m0.024s	
8388608		real 0m0.118s user 0m0.084s sys 0m0.032s			real 0m0.069s user 0m0.092s sys 0m0.036s	
16777216		real 0m0.204s user 0m0.148s sys 0m0.056s			real 0m0.128s user 0m0.160s sys 0m0.088s	
33554432		real 0m0.371s user 0m0.248s sys 0m0.120s			real 0m0.263s user 0m0.312s sys 0m0.204s	
67108864		real 0m0.370s user 0m0.240s sys 0m0.128s			real 0m0.242s user 0m0.344s sys 0m0.128s	