

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): Néstor Rodríguez Vico

Grupo de prácticas: A1

Fecha de entrega:

Fecha evaluación en clase: 7/4

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 bucle\n",
omp_get_thread_num(), i);
    }
    return(0);
}
```

RESPUESTA: código fuente `sectionsModificado.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
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(int argc, char ** argv) {
    #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.

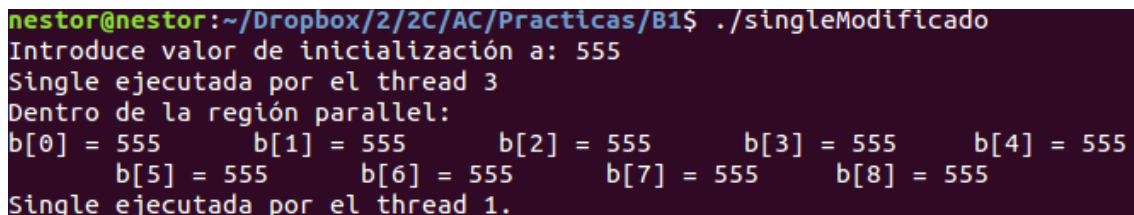
RESPUESTA: código fuente `singleModificado.c`

```

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main(int argc, char ** argv){
    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("Dentro 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());
        }
    }
    return 0;
}

```

CAPTURAS DE PANTALLA:



```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ ./singleModificado
Introduce valor de inicialización a: 555
Single ejecutada por el thread 3
Dentro de la región parallel:
b[0] = 555    b[1] = 555    b[2] = 555    b[3] = 555    b[4] = 555
b[5] = 555    b[6] = 555    b[7] = 555    b[8] = 555
Single ejecutada por el thread 1.

```

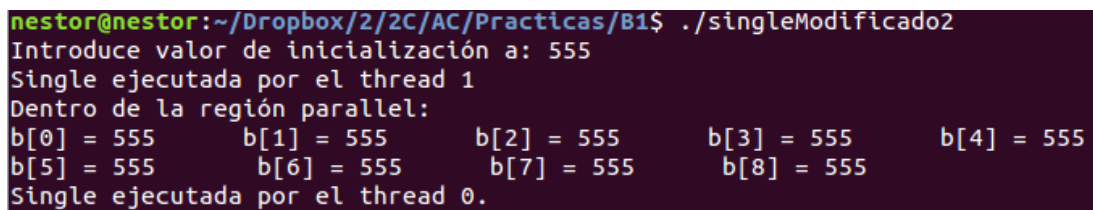
3. Imprimir los resultados del programa `single.c` usando una directiva `master` dentro de la construcción `parallel` en lugar de imprimirlos fuera de la región `parallel`. Añadir lo necesario, dentro de la nueva directiva `master` incorporada, para que se imprima el identificador del thread que ejecuta 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 <stdlib.h>
#include <omp.h>
int main(int argc, char ** argv){
    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("Dentro 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());
        }
    }
    return 0;
}
```

CAPTURAS DE PANTALLA:



```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ ./singleModificado2
Introduce valor de inicialización a: 555
Single ejecutada por el thread 1
Dentro de la región parallel:
b[0] = 555    b[1] = 555    b[2] = 555    b[3] = 555    b[4] = 555
b[5] = 555    b[6] = 555    b[7] = 555    b[8] = 555
Single ejecutada por el thread 0.
```

RESPUESTA A LA PREGUNTA:

La hebra que ejecuta la sección master siempre es la hebra Master (hebra con identificador 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 la directiva `master` no tiene barrera implícita al final. Si no estuviese la directiva `barrier`, la hebra master podría ejecutar el código que imprime la suma calculada antes de que todas las hebras hayan sumado su suma parcial a la suma total.

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 dinámicos**. 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:

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ time `./4-SumaVectores 10000000 `
Tiempo(seg.):0.060984707: no se encontró la orden

real    0m0.811s
user    0m0.212s
sys     0m0.068s

```

RESPUESTA: La suma de CPU del usuario y del sistema es menor que el tiempo real. Esto puede deberse a cualquier cosa, por ejemplo que el proceso haya estado suspendido. El tiempo real mide el tiempo desde que se lanza la ejecución hasta que se termina mientras que la suma del tiempo de usuario más el tiempo del sistema indica el tiempo de ejecución.

6. Generar el código ensamblador a partir del programa secuencial C del Listado 1 para **vectores dinámicos** (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:

```

[A1estudiante23@atcgrid P1]$ cat STDIN.o31051
Tiempo(seg.):0.000000222 / 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) /

[A1estudiante23@atcgrid P1]$ cat STDIN.o31050
Tiempo(seg.):0.088323171 / Tamaño Vectores:10000000 / V1[0]+V2[0]=V3
[0](1000000.000000+1000000.000000=2000000.000000) / / V1[9999999]+V2[9999999]=V3
[9999999](1999999.900000+0.100000=2000000.000000) /

```

RESPUESTA: Cálculo de MIPS

Dentro del bucle (.L9) hay 6 instrucciones. Cuando $n=10$, hay 60 instrucciones más una instrucción antes del bucle más dos instrucciones fuera del bucle. Esto nos da un total de 63 instrucciones para el cálculo de la suma. El tiempo de ejecución para $n=10$ es 0.000000152. Por lo tanto: $MIPS = NI / (T_{CPU} * 10^6) = 63 / 0.000000152 * 10^6 = 414.473684211$ MIPS.

Cuando $n=10000000$, hay 60 instrucciones más una instrucción antes del bucle más dos instrucciones fuera del bucle. Esto nos da un total de 60000003 instrucciones para el cálculo de la suma. El tiempo de ejecución para $n=10000000$ es 0.088323171. Por lo tanto: $MIPS = NI / (T_{CPU} * 10^6) = 60000003 / 0.088323171 * 10^6 = 679251672$ MIPS.

RESPUESTA: Cálculo de MFLOPS

Dentro del bucle (.L9) hay 3 operaciones en coma flotante. Cuando $n=10$, hay 30 operaciones en coma flotante. El tiempo de ejecución para $n=10$ es 0.000000152. Por lo tanto: $MFLOPS = N_{oper-flot} / (T_{CPU} * 10^6) = 30 / 0.000000152 * 10^6 = 197.368421053$ MFLOPS.

Cuando $n=10000000$, hay 30000000 operaciones en coma flotante. El tiempo de ejecución para $n=10000000$ es 0.088323171. Por lo tanto: $\text{MFLOPS} = N_{\text{operflot}} / (T_{\text{CPU}} * 10^6) = 30000000 / (0.047919909 * 10^6) = 626.044594534$ MFLOPS.

código ensamblador generado de la parte de la suma de vectores

```
call    clock_gettime
        xorl    %eax, %eax
        .p2align 4,,10
        .p2align 3
.L9:
        movsd   0(%rbp,%rax,8), %xmm0
        addsd   (%r12,%rax,8), %xmm0
        movsd   %xmm0, (%r15,%rax,8)
        addq    $1, %rax
        cmpl    %eax, %r14d
        ja      .L9
.L10:
        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

```
/* 7-SumaVectoresFor.c
Suma de dos vectores: v3 = v1 + v2
Para compilar usar (-lrt: real tiempo library):
gcc -O2 7-SumaVectoresFor.c -o 7-SumaVectoresFor -lrt
gcc -O2 -S 7-SumaVectoresFor.c -lrt //para generar el código ensamblador
Para ejecutar use: 7-SumaVectoresFor longitud
*/
#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 que imprime todos los
// componentes

// Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de
// los ...
// tres defines siguientes puede estar descomentado):
```

```

// #define VECTOR_LOCAL // descomentar para que los vectores sean variables ...
                        // locales (si se supera el tamaño de la pila se ...
                        // generará el error "Violación de Segmento")

// #define VECTOR_GLOBAL // descomentar para que los vectores sean
variables ...
                        // globales (su longitud no estará limitada por el ...
                        // tamaño de la pila del programa)

#define VECTOR_DYNAMIC // descomentar para que los vectores sean variables ...
                        // dinámicas (memoria reutilizable durante la
ejecución)

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

int main(int argc, char** argv){
    int i;
    double inicio, final, tiempo; // 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)
    #ifdef VECTOR_LOCAL
        // Tamaño variable local en tiempo de ejecución disponible en C a partir de
        actualización C99
        double v1[N], v2[N], v3[N];
    #endif

    #ifdef VECTOR_GLOBAL
        if (N > MAX) N = MAX;
    #endif

    #ifdef VECTOR_DYNAMIC
        double *v1, *v2, *v3;
        v1 = (double*) malloc(N*sizeof(double)); // malloc necesita el tamaño en
        bytes
        v2 = (double*) malloc(N*sizeof(double)); // si no hay espacio suficiente
        malloc devuelve NULL
        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
}

```

```

#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<N; i++){
        v1[i] = N*0.1+i*0.1;
        v2[i] = N*0.1-i*0.1;
    }
}
inicio = omp_get_wtime();

#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<N; i++)
        v3[i] = v1[i] + v2[i];
}
final = omp_get_wtime();
tiempo = final - inicio;

//Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
printf("%f\t%f\t%f\t\n",tiempo, v3[0], v3[N-1]);
/*printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",tiempo,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(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",tiempo,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
#endif

if (N < 100)
    for(i=0; i<N; i++)
        printf("%f + %f = %f\n",v1[i], v2[i], v3[i]);

#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;
}

```

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ gcc 7-SumaVectoresParalelizado.c -o
7-SumaVectoresParalelizado -fopenmp
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ ./7-SumaVectoresParalelizado 8
Tiempo(seg.):0.000006686 / Tamaño Vectores:8 / V1[0]+V2[0]=V3[0](0.800
000+0.800000=1.600000) / / V1[7]+V2[7]=V3[7](1.500000+0.100000=1.600000) /
0.800000 + 0.800000 = 1.600000
0.900000 + 0.700000 = 1.600000
1.000000 + 0.600000 = 1.600000
1.100000 + 0.500000 = 1.600000
1.200000 + 0.400000 = 1.600000
1.300000 + 0.300000 = 1.600000
1.400000 + 0.200000 = 1.600000
1.500000 + 0.100000 = 1.600000
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ ./7-SumaVectoresParalelizado 11
Tiempo(seg.):0.000006159 / Tamaño Vectores:11 / V1[0]+V2[0]=V3[0](1.100
000+1.100000=2.200000) / / V1[10]+V2[10]=V3[10](2.100000+0.100000=2.200000) /
1.100000 + 1.100000 = 2.200000
1.200000 + 1.000000 = 2.200000
1.300000 + 0.900000 = 2.200000
1.400000 + 0.800000 = 2.200000
1.500000 + 0.700000 = 2.200000
1.600000 + 0.600000 = 2.200000
1.700000 + 0.500000 = 2.200000
1.800000 + 0.400000 = 2.200000
1.900000 + 0.300000 = 2.200000
2.000000 + 0.200000 = 2.200000
2.100000 + 0.100000 = 2.200000

```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

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

```

/* 8-SumaVectoresSections.c
Suma de dos vectores: v3 = v1 + v2
Para compilar usar (-lrt: real tiempo library):
gcc -O2 8-SumaVectoresSections.c -o 8-SumaVectoresSections -lrt
gcc -O2 -S 8-SumaVectoresSections.c -lrt //para generar el código
ensamblador
Para ejecutar use: 8-SumaVectoresSections longitud
*/
#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 que imprime todos los
componentes

```



```

//Sólo puede estar definida una de las tres constantes VECTOR_ (sólo uno de
los ...
//tres defines siguientes puede estar descomentado):

//#define VECTOR_LOCAL // descomentar para que los vectores sean variables ...
// locales (si se supera el tamaño de la pila se ...
// generará el error "Violación de Segmento")

//#define VECTOR_GLOBAL // descomentar para que los vectores sean
variables ...
// globales (su longitud no estará limitada por el ...
// tamaño de la pila del programa)

#define VECTOR_DYNAMIC // descomentar para que los vectores sean variables ...
// dinámicas (memoria reutilizable durante la
ejecución)

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

int main(int argc, char** argv){
    //Indices para cada hebra
    int i, j, k, l;
    double inicio, fin, tiempo;
    //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)
    #ifdef VECTOR_LOCAL
        // Tamaño variable local en tiempo de ejecución disponible en C a partir de
        actualización C99
        double v1[N], v2[N], v3[N];
    #endif

    #ifdef VECTOR_GLOBAL
        if (N>MAX) N=MAX;
    #endif

    #ifdef VECTOR_DYNAMIC
        double *v1, *v2, *v3;
        v1 = (double*) malloc(N*sizeof(double)); // malloc necesita el tamaño en
        bytes
        v2 = (double*) malloc(N*sizeof(double)); //si no hay espacio suficiente
        malloc devuelve NULL
        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

    printf("Numero de hebras: %d\n", omp_get_max_threads());

    //Inicializacion dividida en cuatro partes para aprovechar las 4 hebras.

```

```

#pragma omp parallel sections
{
    #pragma omp section
    {
        for(i=0; i<N/2; i++){
            v1[i] = N*0.1+i*0.1;
        }
    }
    #pragma omp section
    {
        for(j=0; j<N/2; j++){
            v2[j] = N*0.1-j*0.1;
        }
    }
    #pragma omp section
    {
        for(k=N/2; k<N; k++){
            v1[k] = N*0.1+k*0.1;
        }
    }
    #pragma omp section
    {
        for(l=N/2; l<N; l++){
            v2[l] = N*0.1-l*0.1;
        }
    }
}

inicio = omp_get_wtime();
//Calcular suma de vectores aprovechando tambien las 4 hebras

#pragma omp parallel sections
{
    #pragma omp section
    {
        for(i=0; i<N/4; i++){
            v3[i] = v1[i] + v2[i];
        }
    }
    #pragma omp section
    {
        for(j=N/4; j<N/2; j++){
            v3[j] = v1[j] + v2[j];
        }
    }
    #pragma omp section
    {
        for(k=N/2; k<3*N/4; k++){
            v3[k] = v1[k] + v2[k];
        }
    }
    #pragma omp section
    {
        for(l=3*N/4; l<N; l++){
            v3[l] = v1[l] + v2[l];
        }
    }
}

fin = omp_get_wtime();
tiempo = fin - inicio;

```

```

//Imprimir resultado de la suma y el tiempo de ejecución
#ifdef PRINTF_ALL
printf("%f\t%f\t%f\t\n",tiempo, v3[0], v3[N-1]);
/*
printf("Tiempo(seg.):%11.9f\t / Tamaño Vectores:%u\n",tiempo,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(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",tiempo,N,v1[0],v2[0],v3[0],N-1,N-1,N-1,v1[N-1],v2[N-1],v3[N-1]);
#endif

if (N < 100)
    for(i=0; i<N; i++)
        printf("%f + %f = %f\n",v1[i], v2[i], v3[i]);

#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;
}

```

(RECUERDE ADJUNTAR CÓDIGO FUENTE AL .ZIP)

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

```

nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ gcc 8-SumaVectores.c -o 8-SumaVectores -fopenmp
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ ./8-SumaVectores 8
Numero de hebras: 4
Tiempo(seg.):0.003831260      / Tamaño Vectores:8      / V1[0]+V2[0]=V3[0](0.800000+0.800000=1.600000)
) / / V1[7]+V2[7]=V3[7](1.500000+0.100000=1.600000) /
0.800000 + 0.800000 = 1.600000
0.900000 + 0.700000 = 1.600000
1.000000 + 0.600000 = 1.600000
1.100000 + 0.500000 = 1.600000
1.200000 + 0.400000 = 1.600000
1.300000 + 0.300000 = 1.600000
1.400000 + 0.200000 = 1.600000
1.500000 + 0.100000 = 1.600000
nestor@nestor:~/Dropbox/2/2C/AC/Practicas/B1$ ./8-SumaVectores 11
Numero de hebras: 4
Tiempo(seg.):0.000002600      / Tamaño Vectores:11      / V1[0]+V2[0]=V3[0](1.100000+1.100000=2.200000)
) / / V1[10]+V2[10]=V3[10](2.100000+0.100000=2.200000) /
1.100000 + 1.100000 = 2.200000
1.200000 + 1.000000 = 2.200000
1.300000 + 0.900000 = 2.200000
1.400000 + 0.800000 = 2.200000
1.500000 + 0.700000 = 2.200000
1.600000 + 0.600000 = 2.200000
1.700000 + 0.500000 = 2.200000
1.800000 + 0.400000 = 2.200000
1.900000 + 0.300000 = 2.200000
2.000000 + 0.200000 = 2.200000
2.100000 + 0.100000 = 2.200000

```

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

RESPUESTA: Para el ejercicio 7 (versión for) el número máximo de hebras que pueden ejecutar el código es igual al número de componentes que tiene el vector. En este caso se asignaría una componente a cada hebra.

Para el ejercicio 8 (versión sections) el número máximo de hebras que pueden ejecutar el código es el número de sections que tenemos, en mi caso 4.

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

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.

Local:

Nº de Componentes	T. secuencial vect. Globales 1thread/core	T. paralelo (versión for) 4threads/cores	T. paralelo (versión sections) 4threads/cores
16384	0.000207	0.000207	0.003463
32768	0.000286	0.000286	0.007976
65536	0.000549	0.000549	0.001263
131072	0.001035	0.001035	0.000371
262144	0.003844	0.003844	0.000796
524288	0.003198	0.003198	0.002463
1048576	0.005499	0.005499	0.004300
2097152	0.009155	0.009155	0.013757
4194304	0.027284	0.027284	0.024959
8388608	0.030730	0.030730	0.036245
16777216	0.060878	0.060878	0.068170
33554432	0.128877	0.128877	0.144453
67108864	0.248178	0.248178	0.281208

atcgrid

Nº de Componentes	T. secuencial vect. Globales 1thread/core	T. paralelo (versión for) 24threads/cores	T. paralelo (versión sections) 24threads/cores
16384	0.000091	0.001701	0.004206
32768	0.000207	0.000082	0.000090
65536	0.000389	0.000098	0.000175
131072	0.000820	0.000181	0.000403
262144	0.001369	0.000348	0.000787
524288	0.002929	0.000485	0.001478
1048576	0.005003	0.000937	0.003042
2097152	0.009982	0.004540	0.007089
4194304	0.019856	0.004926	0.011086
8388608	0.039410	0.011201	0.023649
16777216	0.078878	0.018549	0.042307
33554432	0.157662	0.037246	0.074856
67108864	0.315513	0.073146	0.123510

código que imprima todos los componentes del resultado.

RESPUESTA:

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 el caso de las versiones en paralelo si es mayor, por el tiempo de overhead (creación y destrucción de hebras, asignación de tareas a hebras, etc).

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.

Local

Tamaño	Secuencial – Dinamicos (1 thread/core)			Paralelo – For (4 threads/cores)		
	elapsed	user	sys	elapsed	user	sys
16384	0,001	0,000	0,000	0,001	0,001	0,000
32768	0,001	0,000	0	0,001	0,001	0,000
65536	0,001	0,001	0,000	0,001	0,000	0,001
131072	0,002	0,001	0,001	0,001	0,002	0,001
262144	0,003	0,002	0,001	0,002	0,004	0,001
524288	0,007	0,007	0	0,01	0,020	0,010
1048576	0,013	0,007	0,005	0,007	0,018	0,008
2097152	0,025	0,012	0,012	0,022	0,050	0,024
4194304	0,049	0,031	0,018	0,035	0,09	0,035
8388608	0,097	0,046	0,051	0,056	0,138	0,073
16777216	0,194	0,102	0,091	0,107	0,248	0,16
33554432	0,389	0,215	0,173	0,199	0,471	0,302
67108864	0,778	0,403	0,373	0,389	0,904	0,613

Atcgrid

Tamaño	Secuencial – Dinamicos (1 thread/core)			Paralelo – For (24 threads/cores)		
	elapsed	user	sys	elapsed	user	sys
16384	0,005	0,000	0,002	0,019	0,225	0,001
32768	0,002	0,001	0,001	0,002	0,000	0,016
65536	0,003	0,000	0,003	0,009	0,139	0,001
131072	0,004	0,001	0,003	0,004	0,025	0,046
262144	0,006	0,002	0,004	0,015	0,201	0,002
524288	0,01	0,004	0,006	0,011	0,176	0,017
1048576	0,019	0,01	0,009	0,008	0,108	0,036
2097152	0,037	0,017	0,02	0,013	0,131	0,082
4194304	0,072	0,028	0,044	0,029	0,304	0,115
8388608	0,141	0,061	0,08	0,043	0,344	0,275
16777216	0,278	0,122	0,151	0,074	0,549	0,524
33554432	0,554	0,232	0,317	0,142	0,837	0,959
67108864	1,103	0,457	0,635	0,276	1,523	2,26