CURSO 25/26
PROGRAMACIÓN DE ARQUITECTURAS MULTINÚCLEO
GRADO EN INGENIERÍA INFORMÁTICA, 4TO. CURSO
FACULTAD DE INFORMÁTICA. UNIVERSIDAD DE MURCIA

Trabajo práctico 2

Programación con OpenMP

Alejandro Tomás Martínez (alejandro.tomasm@um.es)

Índice

1. Ejercicio 1

Razona el comportamiento del `ejemplo_for.c`, al ejecutarlo, con 16 hilos, usando distintos `schedules(static, dynamic, guided)` y tamaños de reparto (1 y 2).

Al tener dieciséis hilos, el total de iteraciones del bucle for será de $16 \times 2 = 32$ debido a la condición de parada del bucle:

$$for(i = 0; i < (np * 2); i++)$$

- Con el **planificador estático**, los grupos de trabajo se asignan por igual a todos los hilos. Al ser el tamaño de dicho grupo 1, todos los hilos se reparten todas las iteraciones por igual, tocando a 2 iteraciones por hilo. En el caso de tener dos iteraciones por grupo, sucede lo mismo, puesto que a cada hilo ya le tocaban dos iteraciones.

- Al usar un **planificador dinámico**, el trabajo se asigna según vayan terminando los hilos la carga de trabajo. Con una unidad de trabajo, es muy posible que los hilos que primero entren a ejecutar el bucle reciban casi todo el trabajo, puesto que terminan su trabajo asignado muy rápido. Esto hace que la gran mayoría de hilos no tenga trabajo.

  Al aumentar a dos el número de iteraciones por carga de trabajo, la carga de trabajo se reparte entre más hilos, puesto que teóricamente, realizar dos iteraciones es el doble de costoso que realizar solo una.

- El **planificador guiado** trabaja como un planificador dinámico, pero va reduciendo el número de instrucciones de cada grupo de trabajo desde $\frac{numIteracionesRestantes}{numHilos}$ hasta $tamTrabajo$.

  Dada esta lógica, empleando un tamaño de trabajo de una unidad, el primer hilo ejecutará $\frac{32}{16} = 2$, el siguiente $\frac{30}{16} = 1,875 = 1$ (el algoritmo trunca el resultado) y de igual manera el resto de hilos que ejecuten las iteraciones. Para un tamaño de trabajo de dos, como la primera división ya es igual al tamaño de trabajo, funcionará igual que un algoritmo dinámico convencional.

2. Ejercicio 2

Compara razonadamente el comportamiento de `ejemplo_sections.c` vs. `ejemplo_single.c`. En el caso de *single*, cada macro single tiene una barrera implícita de sincronización al final, por lo que tras cada `printf`, al tener que hacer un `sleep` habrá que esperar antes de salir del bloque single para comenzar el siguiente, haciendo que la ejecución sea secuencial.

Con la macro *sections*, puesto que si bien cada sección es también ejecutada por un solo hilo, todos los section pueden ejecutarse simultáneamente (si hay un número de hilos igual o superior), y solo se sincronizarán al final del constructor, dando lugar a una ejecución paralela.

3. Ejercicio 3

Compara razonadamente el comportamiento de `ejemplo_sections.c` vs. `ejemplo_critical.c`.

En el caso del constructor *critical*, cada uno de los hilos ejecutará el código contenido en su interior, de uno en uno. En el caso de *sections*, todas las secciones se ejecutarán de manera paralela.

4. Ejercicio 4

Compara los programas `ejemplo_critical_pi.c` y `ejemplo_atomic_pi.c` ¿Ofrecen el mismo resultado? ¿Qué ocurre con sus tiempos de ejecución conforme aumenta el número de hilos que se utilizan? Razona tus respuestas.

La siguiente figura muestra el tiempo de ejecución para sendos algoritmos, en base al número de hilos utilizados:

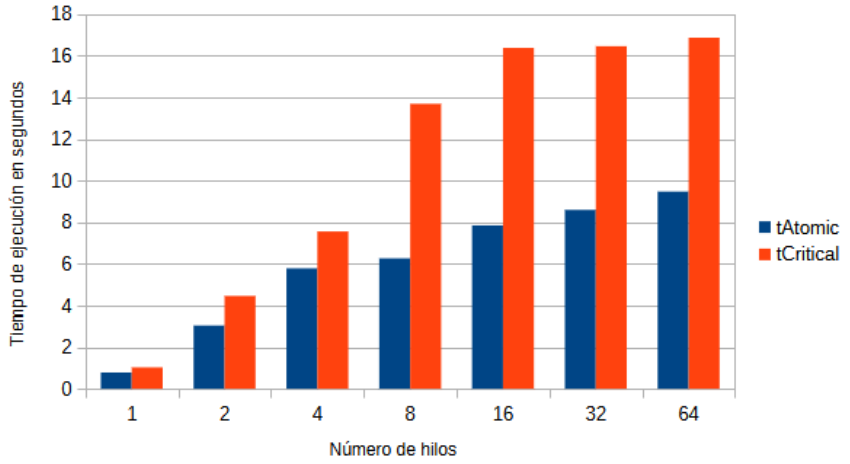

Figura 1: Comparativa entre los tiempos de ejecución de `ejemplo_atomic_pi` y `ejemplo_critical_pi`

Con respecto al resultado que dan, ofrecen una precisión similar, por lo que ofrecen un resultado que puede considerarse "el mismo".

En el aspecto del rendimiento, es similar para un número de hilos inferior a 4. A partir de esta cantidad la penalización en el rendimiento del código que utiliza se dispara, siendo el doble de costoso que con atomic, sin embargo a partir de los 16 hilos el rendimiento se estabiliza en ambos algoritmos, debido seguramente a que hemos superado el número de hilos físicos del procesador, que es de 12, no existiendo mayor paralelismo que pueda verse influenciado por las directivas.

Este comportamiento está justificado, dado que la directiva atomic hace uso de hardware específico en el procesador para realizar operaciones atómicas, permitiendo la ejecución de una instrucción atómica por parte de varios hilos. No obstante, directiva critical es más genérica; detiene la ejecución para otros hilos mientras hay uno dentro

del bloque y conlleva un mayor sobrecoste, además de que si hubiera otra región crítica en otra zona del código, estas compartirían el semáforo o cerrojo que impide su acceso, incurriendo en una degradación del rendimiento notable.

5. Ejercicio 5

En el programa `ejemplo_flush_2.c`, ¿para qué se utiliza la variable `flag`? Muestra y razona el comportamiento al ejecutarse si no se utiliza esta variable.

La variable `flag` tiene como objetivo indicar al segundo hilo de ejecución la lectura de un dato en la variable `data`, de tal forma que el segundo hilo pueda leerla, en este orden.

En el supuesto en el que no exista la variable de sincronización, ambos hilos de ejecución se superpondrán, por lo que el segundo no espera la lectura de datos del primero, y muestra un valor erróneo leído de memoria sin inicializar.

Terminal: Resultado del programa `ejemplo_flush_2.c` con variable flag.

```
host@alex:#/codigo/T_2_OpenMP_ejemplos$ ./programa
Thread 0, esperando dato tecleado por usuario:
ANTES: Thread 1, dato a procesar = 0. Esperando dato desde el otro
↪  thread..
1
Thread 0, ha puesto en común el dato tecleado por el usuario: 1
DESPUES: Thread 1: dato a procesar = 1
```

Terminal: Resultado del programa `ejemplo_flush_2.c` sin la variable flag.

```
host@alex:#/codigo/T_2_OpenMP_ejemplos$ ./programa
Thread 0, esperando dato tecleado por usuario:
ANTES: Thread 1, dato a procesar = 32737. Esperando dato desde el otro
↪  thread..
DESPUES: Thread 1: dato a procesar = 32737
5
Thread 0, ha puesto en común el dato tecleado por el usuario: 5
```

6. Ejercicio 6

A partir de `ejemplo_flush_2.c`, crea `ejercicio_flush_3.c`, donde un hilo lee 2 datos numéricos desde el teclado, un segundo hilo los suma y un tercer hilo muestra el resultado.

La implementación utiliza el mismo mecanismo de sincronización que el ejemplo de referencia, utilizando dos variables como flag para realizar una espera activa para la

lectura y la suma de las variables antes de mostrar estas por el terminal.

ejercicio_flush_3.c

```
int main()
{
    int d1;
    int d2;
    int res;
    int flag2=0;
    int flag=0;
    int iam;
    #pragma omp parallel sections num_threads(3)  private(iam)
    {
    #pragma omp section
    {
        iam=omp_get_thread_num();
        printf("Thread %d, esperando primer dato tecleado por usuario:
        ↪  \n",iam);
        scanf("%d",&d1);
        printf("Thread %d, esperando segundo dato tecleado por usuario:
        ↪  \n",iam);
        scanf("%d",&d2);

        #pragma omp flush(d1,d2)
        flag = 1;
        #pragma omp flush(flag)

        printf("Thread %d, ha puesto en común los datos: %d y %d\n",
        ↪  iam, d1, d2);
    }
    #pragma omp section
    {
        iam=omp_get_thread_num();
        printf("Hilo %d esperando para sumar.\n", iam);
        while (!flag)
        {
            #pragma omp flush(flag)
        }
        #pragma omp flush(d1,d2)

        res = d1 + d2;
        flag2 = 1;
        printf("Thread %d, sumando %d y %d notificados por flag.\n",
        ↪  iam, d1, d2);
    }
```

ejercicio_flush_3.c (2)

```
    #pragma omp section
    {
        iam=omp_get_thread_num();
        printf("Hilo %d esperando para mostrar resultado.\n", iam);

        while (!flag2)
        {
            #pragma omp flush(flag2)
        }
        #pragma omp flush(res)
        printf("Thread: %d, mostrando el resultado de la operación:
        ↪ %d\n", iam, res);
    }
    } //parallel
}
```

Terminal: Resultado del programa implementado para el ejercicio 6.

```
host@alex:#/codigo/T_2_OpenMP_ejemplos$ ./ejemplo_flush_2
Thread 1, esperando primer dato tecleado por usuario:
Hilo 0 esperando para sumar.
Hilo 2 esperando para mostrar resultado.
10
Thread 1, esperando segundo dato tecleado por usuario:
5
Thread 1, ha puesto en común los datos: 10 y 5
Thread: 2, mostrando el resultado de la operación: 15
Thread 0, sumando 10 y 5 notificados por flag.
host@alex:#/codigo/T_2_OpenMP_ejemplos$
```

7. Ejercicio 7

Ejecuta varias veces el programa `ejemplo_lastprivate.c`. ¿Muestra siempre el mismo resultado de la variable x en el último mensaje en pantalla: *"Después de pragma parallel x=..."*? Cambia el `schedule` del bucle for a `dynamic` y repite el ejercicio. Razona tus respuestas.

Con un planificador estático, el total de iteraciones del bucle es repartido de antemano entre los hilos. Como la cláusula lastprivate asigna como valor final el de la última iteración y esta es asignada siempre al mismo hilo, el resultado final de la variable x es siempre el mismo.

Sin embargo, al utilizar un planificador dinámico, los hilos van pidiendo bajo demanda las iteraciones según las van ejecutando, por lo que la iteración final le será asignada a un hilo diferente cada vez, dando lugar a un resultado distinto entre ejecuciones del

programa.

8. Ejercicio 8

Modifica el programa `ejemplo_reduction.c` para que en lugar de calcular la suma de los valores de la variable x de todos los hilos, muestre el valor máximo de dichos valores.

Podemos modificar la cláusula `reduction`, estableciendo como operación aplicada sobre x aquella que almacene el mayor de los valores obtenidos por los distinto hilos con el identificador de reducción `max`.

ejemplo_reduction_mod.c

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

int main()
{
    int x=1000;
    int iam,np,i,j;
    int xmax;

    printf("\n Antes de pragma parallel x=%d \n\n",x);

    #pragma omp parallel private(iam,np,i) reduction(max:x)
    {
        #if defined (_OPENMP)
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        #endif

        printf("Soy el thread %d, antes de actualizar, con x=%d
        ↪  \n",iam,x);
        x=iam*10+1000;
        printf("\t\tSoy el thread %d, despues de actualizar, con x=%d
        ↪  \n",iam,x);

    }//parallel

    printf("\n Despues de pragma parallel x=%d \n\n",x);
}//main
```

9. Ejercicio 9

En el programa `ejemplo_nested.c`, ¿qué ocurre si cambias `omp_set_dynamic(0)` por `omp_set_dynamic(1)`? Muestra el comportamiento al ejecutarlo y razona tu respuesta. ¿Y si cambias `omp_nested(1)` por `omp_nested(0)`? Muestra el comportamiento al ejecutarlo y razona tu respuesta.

En el caso de modificar `omp_set_dynamic`, establecer el valor a 1 no resulta en ningún cambio en el comportamiento del programa: la primera región paralela, debido a la directiva `omp_set_num_threads()` se ejecuta con dos hilos, que a su vez emplean nuevamente el comando para establecer el número de hilos a 3, por lo que el siguiente constructor *parallel* al ser ejecutado por los hilos estos crearán cada uno un nuevo grupo de trabajo con 3 hilos a su vez dada la segunda directiva para establecer threads.

Ejecución del programa `ejecución_nested` con `omp_set_dynamic(1)`.

```
host@alex:#/codigo/T_2_OpenMP_ejemplos$ ./ejemplo_nested
Hello from thread 2 out of 2
        Numero de threads actuales = 2
Hello from thread 1 out of 2
        Numero de threads actuales = 2
            Hello from thread 1 out of 3
                numero de threads actuales = 3
            Hello from thread 2 out of 3
                numero de threads actuales = 3
            Hello from thread 3 out of 3
                numero de threads actuales = 3
            Hello from thread 1 out of 3
                numero de threads actuales = 3
            Hello from thread 3 out of 3
                numero de threads actuales = 3
            Hello from thread 2 out of 3
                numero de threads actuales = 3
host@alex:#/codigo/T_2_OpenMP_ejemplos$
```

Por otro lado, al establecer `omp_nested` a 0, se deshabilita la creación de nuevos bloques de hilos dentro de hilos, por lo que el segundo constructor `parallel` solo es ejecutado por el hilo que ejecuta el bloque.

Ejecución del programa ejecución_nested con omp_nested(0).

```
host@alex:#/codigo/T_2_OpenMP_ejemplos$ ./ejemplo_nested
Hello from thread 1 out of 2
        Numero de threads actuales = 2
            Hello from thread 1 out of 1
                numero de threads actuales = 1
Hello from thread 2 out of 2
        Numero de threads actuales = 2
            Hello from thread 1 out of 1
                numero de threads actuales = 1
host@alex:#/codigo/T_2_OpenMP_ejemplos$
```

10. Ejercicio 10

A partir del ejemplo ejemplo_atomic_pi.c, crea el programa ejercicio_lock_pi.c que realiza el mismo cálculo pero, en lugar de utilizar operaciones atómicas, haga uso de un candado para sincronizar el acceso en exclusión mutua a la variable sum que es compartida por todos los hilos.

ejercicio_lock_pi.c

```c
#include <omp.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int i;
    int nt, max = 1;
    const long n = 100000000;

    double acum;
    double h, x, pi, sum;
    double pi21 = 3.14159265358979323846;

    #if defined (_OPENMP)
        max = omp_get_max_threads();
    #endif

    if(argc < 2) {
        fprintf(stderr, "Usage: %s <omp_threads>\n", argv[0]);
        return(EXIT_FAILURE);
    }
```

ejercicio_lock_pi.c (2)

```
    nt = (max > 1) ? atoi(argv[1]) : max;

    if(nt < 1) {
        fprintf(stderr, "Invalid Value for OMP_THREADS...\n");
        return(EXIT_FAILURE);
    }

    printf("\n");
    printf("MAX_OMP_THREADS: %d\n", max);
    printf("OMP_THREADS: %d\n", nt);
    printf("\n");

    sum = 0.0;
    h = 1.0 / (double) n;
    // Inicializamos el cerrojo
    omp_lock_t cerrojo;
    omp_init_lock(&cerrojo);

    double ti, tf;
    ti = omp_get_wtime();

    #pragma omp parallel for shared(sum) private(i,x,acum)
    ↪  firstprivate(h) num_threads(nt)
    for(i=1; i <= n; i++) {
        x = h*((double) i - 0.5);
        acum = (4.0/(1.0 + x*x));
        // Para ejecutar la sección crítica debemos poseer el cerrojo
        omp_set_lock(&cerrojo);
        sum += acum;
        omp_unset_lock(&cerrojo);
    }
    tf = omp_get_wtime();
    // Liberamos los recursos del cerrojo.
    omp_destroy_lock(&cerrojo);

    pi = sum*h;
    printf("Time = %.4lf seconds\n", (tf - ti));
    printf("El valor aproximado de PI es: %1.16lf, con un error de
    ↪  %1.16lf\n", pi, fabs(pi - pi21));

    printf("\n");
    return(EXIT_SUCCESS);
}
```

11. Ejercicio 11

A partir del ejemplo de Fibonacci, `ejemplo_fibo.c`, crea la versión `ejercicio_fibo_2.c`, con el objetivo de reducir la profundidad del árbol de tareas OpenMP que se generan. Para ello, cuando el número a calcular sea menor que una cota dada, (por ejemplo, menor que 20) el cálculo se debe hacer directamente en la tarea correspondiente, mediante las oportunas llamadas recursivas secuenciales, pero sin que se generen más tareas OpenMP a partir de ella. Comprueba razonadamente que el programa tiene este comportamiento.

Se ha utilizado la función de la librería `omp_in_final` junto a la cláusula **final()** para poder diferenciar dos casos clave:

- Una región paralela normal, siempre que $n \geq 5$ en la que al ejecutar la cláusula *task* se creará una nueva tarea que podrá ejecutar cualquier hilo del constructor parallel.
- Una región paralela final, cuando $n < 5$, en la que no se creará una nueva tarea, y que será ejecutada de manera secuencial por el mismo hilo que ha entrado a ella. Se puede comprobar si estamos ejecutando esta seción con la función de la librería `omp_in_final` antes mencionada, que devolverá *true* en dicho caso.

Estos dos casos están diferenciados con logs por consola con la etiqueta [PAR] y [SEQ] respectivamente, para poder seguir la ejecución.

ejercicio_fibo_2.c

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

long comp_fib_numbers(int n){
        long fnm1, fnm2, fn;
        int        iam;

        iam=omp_get_thread_num();
        // En una región final, estamos ejecutando en el mismo hilo.
        if (omp_in_final()){
                printf("[SEQ][T%d] Ejecutando fibo(%d).\n", iam, n);
        } else {
                printf("[PAR][T%d] Ejecutando fibo(%d).\n", iam, n);
        }

        if ( n == 0 || n == 1 )
                        return(n);
        // Si el n es menor que 5, se ejecuta en la misma tarea y en
        ↪  una región tipo "final", donde no se crean nuevas tareas en
        ↪  paralelo.
        #pragma omp task shared(fnm1) final(n < 5)
        {
        fnm1 = comp_fib_numbers(n-1);
        }

        #pragma omp task shared(fnm2) final(n < 5)
        {
        fnm2 = comp_fib_numbers(n-2);
        }

        #pragma omp taskwait
        fn = fnm1 + fnm2;

        return(fn);
}
int main(int argc,char *argv[])
{
  int n=8;
  int nthreads;
  long        result;
  int        iam;

  if (argc>1) n=atoi(argv[1]);
```

ejercicio_fibo_2.c (2)

```
  omp_set_num_threads(4);

  #pragma omp parallel shared(nthreads)
  {
    iam=omp_get_thread_num();
    printf("Soy el thread %d\n",iam);

    #pragma omp single nowait
    {
        result = comp_fib_numbers(n);
    } // End of single

  } // End of parallel region

  printf("Serie de Fibonacci para n=%d. El resultado es %ld
  ↪  \n",n,result);
  return 0;
}
```

Terminal: Ejecución del programa del ejercicio 11.

```
host@alex:#/proyectos/practicas-PAM-25-26/p2$ ./dist/ejercicio_fibo_2
↪  10 | head -n 16
Soy el thread 0
[PAR][T0] Ejecutando fibo(10).
[PAR][T0] Ejecutando fibo(8).
[PAR][T0] Ejecutando fibo(6).
[PAR][T0] Ejecutando fibo(4).
[SEQ][T0] Ejecutando fibo(2).
[SEQ][T0] Ejecutando fibo(1).
[SEQ][T0] Ejecutando fibo(0).
[SEQ][T0] Ejecutando fibo(3).
[SEQ][T0] Ejecutando fibo(2).
[SEQ][T0] Ejecutando fibo(1).
[SEQ][T0] Ejecutando fibo(0).
Soy el thread 3
[PAR][T0] Ejecutando fibo(9).
[PAR][T0] Ejecutando fibo(7).
[SEQ][T0] Ejecutando fibo(1).
host@alex:#/proyectos/practicas-PAM-25-26/p2$
```

12. Ejercicio 12

Diseña el programa paralelo, usando OpenMP, `MulMatCua.c` para multiplicar matrices cuadradas de números reales en doble precisión, Cnxn=AnxnBnxn. Las matrices A, B y C se encuentran almacenadas en memoria por filas, con un leading dimension (distancia de almacenamiento entre cada par de elementos consecutivos de una columna) de ldn elementos. Se debe repartir el trabajo a realizar de manera que el cálculo de cada conjunto de F filas consecutivas de la matriz C sea asignado a un hilo de entre los t hilos generados.

`SINTAXIS: MulMatCua <n><ldn><F><t>`

MulMatCua.c

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int F = 1;

void rellenarMatrizCua(double *matrizCuadrada, int dimension){
    for(int i=0; i<dimension*dimension; i++){
        double num = (double) rand()/RAND_MAX;
        matrizCuadrada[i] = num;
    }
}

void multiplicarMatrizCua(double *m1, double *m2, double *mRes, int
↪   ldn){
    for(int i=0; i<ldn; i++){
        for(int j=0; j<ldn; j++){
            for(int k=0; k<ldn; k++){
                mRes[i*ldn + j] += m1[i*ldn + k] * m2[k*ldn + j];
            }
        }
    }
}
```

MulMatCua.c (2)

```
void multiplicarMatrizCuaParalelo(double *m1, double *m2, double *mRes
↪  , int ldn){
// Sched static -> Cada hilo recibe una carga de F filas.
// Índices i y j privados por defecto, matrices compartidas.
    #pragma omp parallel for schedule(static, F) shared(m1, m2, mRes)
    for(int i=0; i<ldn; i++){
        for(int j=0; j<ldn; j++){
            for(int k=0; k<ldn; k++){
                //mRes[i,j] = m1[i][k] * m2[k][j];
                mRes[i*ldn + j] += m1[i*ldn + k] * m2[k*ldn + j];
            }
        }
    }
}

int compararMatrices(double * m1, double * m2, int dimension) {
    int res = 1;
    for(int i=0;i<dimension*dimension;i++){
        if(m1[i]!=m2[i]){
            res = 0;
        }
    }
    return res;
}

void mostrarMatriz(double *m, int dimension){
    for(int i=0; i<dimension; i++){
        for(int j=0; j<dimension; j++){
            printf("%f ", m[i*dimension+j]);
        }
        printf("\n");
    }
}

int main(int argc,char *argv[])
{
    if(argc != 5) {
        printf("Uso: ./programa <n> <ldn> <filasPorHilo> <hilos>\n");
        return EXIT_SUCCESS;
    }
    srand ( (unsigned)time ( NULL ) );

    int n = atoi(argv[1]);
    int ldn = atoi(argv[2]);
    F = atoi(argv[3]);
    int hilos = atoi(argv[4]);
```

MulMatCua.c (3)

```
    omp_set_num_threads(hilos);

    double * A = malloc(sizeof(double) * n * n);
    double * B = malloc(sizeof(double) * n * n);
    double * C = calloc(sizeof(double),  n * n);
    double * D = calloc(sizeof(double),  n * n);

    rellenarMatrizCua(A, n);
    rellenarMatrizCua(B, n);

    double t1, t2;

    t1 = omp_get_wtime();
    multiplicarMatrizCua(A, B, C, ldn);
    t1 = omp_get_wtime() - t1;

    t2 = omp_get_wtime();
    multiplicarMatrizCuaParalelo(A, B, D, ldn);
    t2 = omp_get_wtime() - t2;

    if(compararMatrices(C,D,n)){
        printf("Secuencial y paralelo dan el mismo resultado.\n");
    } else {
        printf("Secuencial y paralelo dan resultados distintos.\n");
    }

    printf("Secuencial: %f.\nParalelo: %f.\n",t1, t2);

    double diff;
    if(t1<t2){
        diff = ((t2/t1) - 1.0) * 100;
        printf("Pérdida de rendimiento del %.2f%%.\n",diff);
    } else {
        diff = ((t1/t2) - 1.0) * 100;
        printf("Ganancia de rendimiento del %.2f%%.\n",diff);
    }

    free(A);
    free(B);
    free(C);
    free(D);

    return EXIT_SUCCESS;
}
```

Terminal: Ejecución del programa del ejercicio 12.

```
host@alex:#/proyectos/practicas-PAM-25-26/p2$ ./dist/MulMatCua 1024
↪  1024 200 12
Secuencial y paralelo dan el mismo resultado.
Secuencial: 2.192585.
Paralelo: 0.427977.
Ganancia de rendimiento del 412.31%.
host@alex:#/proyectos/practicas-PAM-25-26/p2$
```

13. Ejercicio 13

A partir del programa anterior, escribe un programa paralelo usando OpenMP, llamado `MulMat.c`, para calcular el producto de dos matrices rectangulares de números reales de doble precisión, de la forma Cmxn=Amxk Bkxn. Las matrices A, B y C se encuentran almacenadas en memoria por filas, con un leading dimension (distancia de almacenamiento entre cada par de elementos consecutivos de una columna) de lda, ldb y ldc elementos, respectivamente. Se debe repartir el trabajo a realizar de manera que el cálculo de cada conjunto de F filas consecutivas de la matriz C sea asignado a un hilo de entre los t hilos generados.

`SINTAXIS: MulMat <m><n><k><lda><ldb><ldc><F><t>`

MulMat.c

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int F = 1;

void rellenarMatriz(double *m, size_t sz_m){
    for(int i=0; i<sz_m; i++){
        double num = (double) rand()/RAND_MAX;
        m[i] = num;
    }
}
```

MulMat.c (2)

```
void multiplicarMatriz(double *m1, double *m2, double *mRes , int ld1,
↪  int ld2, int ld3, size_t sz_mRes){
    int filas = sz_mRes / ld3;
    for(int i=0; i<filas; i++){
        for(int j=0; j<ld3; j++){
            for(int k=0; k<ld1; k++){
                mRes[i*ld3 + j] += m1[i*ld1 + k] * m2[k*ld2 + j];
            }
        }
    }
}

void multiplicarMatrizParalelo(double *m1, double *m2, double *mRes ,
↪  int ld1, int ld2, int ld3, size_t sz_mRes){
    int filas = sz_mRes / ld3;
    #pragma omp parallel for schedule(static, F) shared(m1, m2, mRes)
    for(int i=0; i<filas; i++){
        for(int j=0; j<ld3; j++){
            for(int k=0; k<ld1; k++){
                mRes[i*ld3 + j] += m1[i*ld1 + k] * m2[k*ld2 + j];
            }
        }
    }
}

int compararMatrices(double * m1, double * m2, size_t sz_m) {
    for(int i=0; i<sz_m; i++){
        if(m1[i]!=m2[i]){
            return 0;
        }
    }
    return 1;
}

void mostrarMatriz(double *m, size_t sz_m, int leading_dim){
    int remain = sz_m;
    int actual = 0;

    while(remain != 0){
        for(int i=0; i<leading_dim; i++){
            printf("%f ", m[actual]);
            remain--;
            actual++;
        }
        printf("\n");
    }
}
```

MulMat.c (3)

```
int main(int argc,char *argv[])
{
    if(argc != 9) {
        printf("Uso: ./MulMat <filasAyC> <columnasByC>
        ↪  <columnasAfilasB> <ldA> <ldB> <ldC> <TrabajoPorHilo>
        ↪  <Hilos>\n");
        return EXIT_SUCCESS;
    }
    srand ( (unsigned)time ( NULL ) );

    int m = atoi(argv[1]);
    int n = atoi(argv[2]);
    int k = atoi(argv[3]);
    int lda = atoi(argv[4]);
    int ldb = atoi(argv[5]);
    int ldc = atoi(argv[6]);
    int ldd = ldc;
    F = atoi(argv[7]);
    int hilos = atoi(argv[8]);
    omp_set_num_threads(hilos);

    size_t sz_a = m * k;
    size_t sz_b = k * n;
    size_t sz_c = m * n;
    size_t sz_d = m * n;
    double * A = malloc(sizeof(double) * sz_a);
    double * B = malloc(sizeof(double) * sz_b);
    double * C = calloc(sz_c, sizeof(double));
    double * D = calloc(sz_d, sizeof(double));

    rellenarMatriz(A, sz_a);
    rellenarMatriz(B, sz_b);

    double t1, t2;

    t1 = omp_get_wtime();
    multiplicarMatriz(A, B, C, lda, ldb, ldc, sz_c);
    t1 = omp_get_wtime() - t1;

    t2 = omp_get_wtime();
    multiplicarMatrizParalelo(A, B, D, lda, ldb, ldd, sz_d);
    t2 = omp_get_wtime() - t2;
```

MulMat.c (4)

```
    if(compararMatrices(C, D, sz_d)){
        printf("Secuencial y paralelo dan el mismo resultado.\n");
    } else {
        printf("Secuencial y paralelo dan resultados distintos.\n");
    }

    printf("Secuencial: %f.\nParalelo: %f.\n",t1, t2);

    double diff;
    if(t1<t2){
        diff = ((t2/t1) - 1.0) * 100;
        printf("Pérdida de rendimiento del %.2f%%.\n",diff);
    } else {
        diff = ((t1/t2) - 1.0) * 100;
        printf("Ganancia de rendimiento del %.2f%%.\n",diff);
    }

    free(A);
    free(B);
    free(C);
    free(D);

    return EXIT_SUCCESS;
}
```

Terminal: Ejecución del programa del ejercicio 13.

```
host@alex:#/proyectos/practicas-PAM-25-26/p2$ ./dist/MulMat 1024 2048
↪  1024 1024 2048 2048 200 12
Secuencial y paralelo dan el mismo resultado.
Secuencial: 14.971638.
Paralelo: 5.027090.
Ganancia de rendimiento del 197.82%.
host@alex:#/proyectos/practicas-PAM-25-26/p2$
```

14. Ejercicio 14

A partir del programa anterior, escribe un programa paralelo usando tareas de OpenMP, llamado `TMulMat.c`, para calcular el producto de dos matrices rectangulares de números reales de doble precisión, de la forma Cmxn =AmxkBkxn. Las matrices A, B y C se encuentran almacenadas en memoria por filas, con un leading dimension (distancia de almacenamiento entre cada par de elementos consecutivos de una columna) de lda, ldb y

ldc elementos, respectivamente. Se debe repartir el trabajo a realizar de manera que el cálculo de cada conjunto de F filas consecutivas de la matriz C sea asignado a una tarea.

SINTAXIS: TMulMat <m><n><k><lda><ldb><ldc><F><t>

TMulMat.c

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int F = 1;

void rellenarMatriz(double *m, size_t sz_m){
    for(int i=0; i<sz_m; i++){
        double num = (double) rand()/RAND_MAX;
        m[i] = num;
    }
}

void multiplicarMatriz(double *m1, double *m2, double *mRes , int ld1,
↪  int ld2, int ld3, size_t sz_mRes){
    int filas = sz_mRes / ld3;
    for(int i=0; i<filas; i++){
        for(int j=0; j<ld3; j++){
            for(int k=0; k<ld1; k++){
                //mRes[i,j] = m1[i][k] * m2[k][j];
                mRes[i*ld3 + j] += m1[i*ld1 + k] * m2[k*ld2 + j];
            }
        }
    }
}
```

TMulMat.c (2)

```
void multiplicarMatrizTareas(double *m1, double *m2, double *mRes , int
↪  ld1, int ld2, int ld3, size_t sz_mRes){
    size_t filas = sz_mRes / ld3;
    size_t fila;
    double *sub_m1 = m1;
    double *sub_mRes = mRes;
    size_t sz_sub_m = F * ld3; // El número de elementos de la
    ↪  submatriz a calcular define el número de filas.

    for(fila = 0; fila+F < filas; fila+=F){
        #pragma omp task shared(m1, m2, mRes) firstprivate(sub_m1,
        ↪  sub_mRes, sz_sub_m, ld1, ld2, ld3)
        {
        multiplicarMatriz(sub_m1, m2, sub_mRes , ld1, ld2, ld3,
        ↪  sz_sub_m);
        }

        sub_m1 += F*ld1;
        sub_mRes += F*ld3;
    }

    // Calcular las últimas filas, ya que la última iteración podría no
    ↪  ser
    // múltiplo de F.
    sz_sub_m = (filas - fila) * ld3;

    // Si no hay filas, tampoco hay elementos a calcular, por lo que la
    ↪  siguiente llamada no hace nada:
    multiplicarMatriz(sub_m1, m2, sub_mRes , ld1, ld2, ld3, sz_sub_m);

}

int compararMatrices(double * m1, double * m2, size_t sz_m) {
    for(int i=0; i<sz_m; i++){
        if(m1[i]!=m2[i]){
            return 0;
        }
    }
    return 1;
}
```

TMulMat.c (3)

```
int main(int argc,char *argv[])
{
    if(argc != 9) {
        printf("Uso: ./MulMat <filasAyC> <columnasByC>
        ↪  <columnasAfilasB> <ldA> <ldB> <ldC> <TrabajoPorHilo>
        ↪  <Hilos>\n");
        return EXIT_SUCCESS;
    }
    srand ( (unsigned)time ( NULL ) );

    int m = atoi(argv[1]);
    int n = atoi(argv[2]);
    int k = atoi(argv[3]);
    int lda = atoi(argv[4]);
    int ldb = atoi(argv[5]);
    int ldc = atoi(argv[6]);
    int ldd = ldc;
    F = atoi(argv[7]);
    int hilos = atoi(argv[8]);

    omp_set_num_threads(hilos);

    size_t sz_a = m * k;
    size_t sz_b = k * n;
    size_t sz_c = m * n;
    size_t sz_d = m * n;

    double * A = malloc(sizeof(double) * sz_a);
    double * B = malloc(sizeof(double) * sz_b);
    double * C = calloc(sz_c, sizeof(double));
    double * D = calloc(sz_d, sizeof(double));

    rellenarMatriz(A, sz_a);
    rellenarMatriz(B, sz_b);

    double t1, t2;

    t1 = omp_get_wtime();
    multiplicarMatriz(A, B, C, lda, ldb, ldc, sz_c);
    t1 = omp_get_wtime() - t1;
```

TMulMat.c (4)

```
// Creamos un grupo de hilos
    #pragma omp parallel
    {
    // Ejecutamos la multiplicación en un solo hilo, que irá creando
    // las tareas que serán asignadas al resto de hilos.
        #pragma omp single nowait
        {
        t2 = omp_get_wtime();
        multiplicarMatrizTareas(A, B, D, lda, ldb, ldd, sz_d);
        t2 = omp_get_wtime() - t2;
        }

    }

    if(compararMatrices(C, D, sz_d)){
        printf("Secuencial y paralelo dan el mismo resultado.\n");
    } else {
        printf("Secuencial y paralelo dan resultados distintos.\n");
    }

    printf("Secuencial: %f.\nParalelo: %f.\n",t1, t2);

    double diff;
    if(t1<t2){
        diff = ((t2/t1) - 1.0) * 100;
        printf("Pérdida de rendimiento del %.2f%%.\n",diff);
    } else {
        diff = ((t1/t2) - 1.0) * 100;
        printf("Ganancia de rendimiento del %.2f%%.\n",diff);
    }

    free(A);
    free(B);
    free(C);
    free(D);

    return EXIT_SUCCESS;
}
```

Terminal: Ejecución del programa del ejercicio 14.

```
host@alex:#/proyectos/practicas-PAM-25-26/p2$ ./dist/MulMat 1024 2048
↪  1024 1024 2048 2048 20 12
Secuencial y paralelo dan el mismo resultado.
Secuencial: 11.930115.
Paralelo: 0.109989.
Ganancia de rendimiento del 10746.65%.
host@alex:#/proyectos/practicas-PAM-25-26/p2$
```