TEC | Tecnológico de Costa Rica

Instituto Tecnológico de Costa Rica
Área Académica de Ingeniería en Computadores
CE 4302 — Arquitectura de Computadores II

Taller 2:
OpenMP

Fecha de asignación:	24 marzo	Fecha de entrega:	31 marzo
Grupo:	1 persona	Profesor:	Luis Barboza Artavia

Shakime Richards Sparks

2018170667

2. Investigación

1. ¿En qué consiste OpenMP?

OpenMP es una interfaz de programación que permite programar en paralelo en entornos de memoria compartida, como las CPU multinúcleo. Proporciona un conjunto de directivas de compilador y una API para programas C, C++ y FORTRAN, que permite a los desarrolladores especificar regiones de código que pueden ejecutarse en paralelo. Al utilizar OpenMP, los desarrolladores pueden aprovechar la potencia de procesamiento disponible en los sistemas multinúcleo, lo que se traduce en una ejecución más rápida y eficiente de sus programas. Las directivas se utilizan para indicar a la biblioteca de ejecución OpenMP cómo paralelizar el código dentro de la región paralela designada [1].

2. ¿Cómo se define la cantidad de hilos en OpenMP?

Según [1], cuando OpenMP encuentra una directiva de región paralela, crea tantos hilos como núcleos de procesamiento haya en el sistema. Por ejemplo, si el sistema tiene dos núcleos de procesamiento, OpenMP crea dos subprocesos para ejecutar la región paralela. Del mismo modo, si el sistema tiene cuatro núcleos de procesamiento, OpenMP crea cuatro subprocesos para ejecutar la región paralela.

Este funcionamiento se denomina ***número de hilos por defecto*** y viene determinado por el número de núcleos de procesamiento disponibles en el sistema. Ahora bien, los programadores también pueden especificar manualmente el número de subprocesos que se crearán utilizando la API OpenMP. Pueden hacerlo estableciendo la cláusula ***num_threads*** en la directiva de región paralela.

3. ¿Cómo se crea una región paralela en OpenMP?

Cuando OpenMP encuentra la directiva ***omp parallel*** la identifica como una región paralela.

```c
#include < stdio.h >

int main(void)
{
    #pragma omp parallel
    {
    printf("Hello, world.\n");
    }

  return 0;
}
```

El hilo original se denominará hilo maestro con thread-id 0. Cuando el hilo maestro llega a esta línea, bifurca hilos adicionales para realizar el trabajo incluido en el bloque siguiendo el comportamiento del #pragma construct. El bloque es ejecutado por todos los hilos en paralelo [2].

4. ¿Cómo se compila un código fuente c para utilizar OpenMP y qué encabezado debe incluirse?

Las funciones OpenMP se incluyen en un header file llamado ***omp.h***. Sistemas linux como dover y foxcroft tienen gcc/g++ instalado con soporte OpenMP. Todo lo que de debe hacer es utilizar la bandera ***-fopenmp*** en la línea de comandos [2]:

```
gcc -fopenmp hellosmp.c  -o  hellosmp
```

5. ¿Cuál función me permite conocer el número de procesadores disponibles para el programa? Realice un *print* con la función con los procesadores de su computadora.

Según [2], utiliza la función omp_get_num_procs().

processors_quantity.c > ...

```c
#include <omp.h>

#include <stdio.h>
#include <stdlib.h>


int main(void)
{
    printf("Processors Quantity: %d\n", omp_get_num_procs());

  return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
jey@jey:~/Documents/Arquitectura2/Taller2$ gcc processors_quantity.c -o processors_quantity -fopenmp
jey@jey:~/Documents/Arquitectura2/Taller2$ ./processors_quantity
Processors Quantity: 8
```

6. ¿Cómo se definen las variables privadas en OpenMP?¿Por qué son importantes?

Estas variables son privadas para cada subproceso, lo que significa que cada subproceso tendrá su propia copia local, no se inicializan y su valor no se mantiene para su uso fuera de la región paralela. Los contadores de iteración de bucle en las construcciones de bucle OpenMP son privados por defecto [2].

Se definen con **#pragma omp parallel private(variable):**

```c
int main (int argc, char *argv[]) {

  int th_id, nthreads;


#pragma omp parallel private(th_id)

  //th_id is declared above.  It is is specified as private; so each
  //thread will have its own copy of th_id
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);
  }
```

Las variables privadas son importantes en OpenMP porque aseguran el comportamiento correcto en un programa paralelo al permitir que cada subproceso tenga su propia copia independiente de una variable, y también brindan un mejor rendimiento al reducir la necesidad de que los subprocesos accedan a las variables compartidas.

7. ¿Cómo se definen las variables compartidas en OpenMP?¿Cómo se deben actualizar valores de variables compartidas?

Una variable compartida es visible y accesible por todos los subprocesos simultáneamente. Por defecto, todas las variables de la región de trabajo compartido son compartidas excepto el contador de iteración del bucle. Pero se pueden declarar explícitamente como **#pragma omp parallel shared(variable)**

Las variables compartidas deben utilizarse con cuidado, ya que pueden provocar condiciones de carrera. Por lo que, OpenMP proporciona varias formas de actualizar variables compartidas de manera segura, dentro de estas están: reducción, bloqueo, atomic, entre otras.

8. ¿Para qué sirve *flush* en OpenMP?

La directiva **#pragma omp flush** se utiliza para garantizar la consistencia entre la memoria temporal de un hilo y la memoria principal del sistema, o entre la vista de memoria de varios hilos [5]. Cuando se utilizan múltiples hilos en un programa OpenMP, cada hilo tiene su propia memoria caché que puede contener valores de las variables compartidas que son diferentes a los de otras cachés. Esto puede causar inconsistencias en el valor de las variables compartidas y errores en el programa. Para resolver este problema, se emplea esta directiva para asegurar que los valores de las variables compartidas sean actualizados y sincronizados entre los hilos antes de continuar con la ejecución.

Dependiendo del tipo de flush, la operación puede ser más o menos fuerte en la sincronización de la memoria. Una "strong flush" garantiza la consistencia entre la

memoria temporal de un hilo y la memoria principal del sistema, mientras que una "weak flush" solo garantiza la consistencia entre la vista de memoria de los hilos.

NOTA: *Considerar que la directiva* `#pragma omp flush` *no garantiza que todas las operaciones anteriores a la directiva estén completas antes de continuar con la ejecución. Por lo tanto, es recomendable utilizar otras directivas de sincronización, como* `#pragma omp barrier` *o* `#pragma omp critical`*, en combinación con* `#pragma omp flush` *para garantizar una sincronización completa y efectiva entre los hilos.*

9. ¿Cuál es el propósito de *pragma omp single*? ¿En cuáles casos debe usarse?

La directiva #**pragma omp single** es utilizada para indicar que un bloque de código debe ser ejecutado por un solo hilo dentro del equipo paralelo [2]. El propósito de esta directiva es garantizar que el bloque de código sea ejecutado por un solo hilo, lo que puede ser útil en situaciones donde solo una tarea debe ser realizada una vez, como por ejemplo inicializar una variable, imprimir un mensaje, leer un archivo.

10. ¿Cuáles son tres instrucciones para la sincronización? Realice una comparación entre ellas donde incluya en cuáles casos se utiliza cada una.

Bloqueo (Locking): OpenMP proporciona una cláusula ***critical*** que permite que una variable compartida sea actualizada de forma segura mediante el bloqueo de acceso a la variable mientras un hilo la actualiza. Esto asegura que ningún otro hilo pueda acceder a la variable al mismo tiempo. Suele utilizarse para proteger datos compartidos de condiciones de carrera [2]. Por ejemplo, para actualizar una variable compartida **`count`** de forma segura en un bucle utilizando un bloqueo, se debe hacer lo siguiente:

```
// Critical
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    #pragma omp critical
    {
        count++;
    }
}
```

Atomic: OpenMP proporciona una cláusula ***atomic*** que permite que una variable compartida sea actualizada de forma segura mediante una operación atómica (como incremento, decremento o intercambio) realizada por un solo hilo a la vez. No hace que toda la sentencia sea atómica; sólo la actualización de memoria es atómica. Un compilador puede utilizar instrucciones de hardware especiales para obtener un mejor rendimiento que cuando se utiliza critical [2]. Por ejemplo, el mismo ejemplo anterior utilizando atomic, se debe hacer lo siguiente:

```
// Atomic
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    #pragma omp atomic
    {
        count++;
    }
}
```

Barrier: OpenMP proporciona una cláusula ***barrier*** que se utiliza para sincronizar los hilos en un equipo, lo que significa que todos los hilos deben alcanzar el mismo punto en el código antes de que se pueda continuar con la ejecución. Esto puede ser útil en situaciones en las que se necesita asegurar que ciertas tareas en paralelo se completen antes de continuar con el resto del programa. De hecho, toda estructura de trabajo compartido tiene una barrera de sincronización implícita al final [2]. Por ejemplo, para calcular la suma de los elementos en un arreglo en paralelo:

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

int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < 10; i++) {
        sum += arr[i];
        printf("Thread %d: sum = %d\n",
         omp_get_thread_num(), sum);
    }

    // Barrier
    #pragma omp barrier
    printf("Final sum = %d\n", sum);

    return 0;
}
```

11. ¿Cuál es el propósito de *reduction* y cómo se define?

OpenMP proporciona una cláusula de reducción permite que una variable compartida sea actualizada de forma segura mediante una operación de reducción específica (como suma, multiplicación, mínimo o máximo) realizada por todos los hilos en paralelo. El resultado final se almacena en una variable privada [4].

Se define con **#pragma omp reduction (operador : sum)**. Por ejemplo, la siguiente directiva de OpenMP actualiza una variable compartida **sum** de forma segura mediante una operación de reducción de suma.

```
// Reduction
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
    sum += array[i];
}
```

3. Análisis

La constante π puede ser calculada mediante una aproximación de la integral $\int_0^1 \frac{4}{1+x^2}\,dx$. El código fuente `pi.c` muestra, de forma secuencial, cómo calcular dicho valor.

Para este primer código debe realizar lo siguiente:

1. Identifique cuáles secciones se pueden paralelizar, así como cuáles variables pueden ser privadas o compartidas. Justifique.

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
int main ()
{
      int i;
      double x, pi, sum = 0.0;
      double start_time, run_time;

      step = 1.0/(double) num_steps;


      start_time = omp_get_wtime();

      for (i=1;i<= num_steps; i++){
          x = (i-0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
      }

      pi = step * sum;
      run_time = omp_get_wtime() - start_time;
      printf("\n pi with %ld steps is %lf in %lf seconds\n ",
        num_steps,pi,run_time);
}
```

En el código dado, la única sección que se puede paralelizar es el bucle for, dado que las otras secciones del código, utilizan solamente el resultado final del bucle.

Privadas:

- **run_time** y **start_time**: Son variables privadas al estar declaradas dentro del ámbito de la función `main()`. Cuando se ejecuta el programa, cada hilo tendrá su propia instancia de estas variables y las actualizará de manera independiente.
- **i**: La variable `i` es una variable de control del bucle y su valor se utiliza para determinar el índice del array, debe almacenarse como una variable privada.
- **pi**: La variable `pi` es una variable de local y no se actualiza en el bucle, por lo que puede almacenarse como una variable privada.

Compartidas:

- **sum**: La variable `sum` es una variable utilizada dentro del bucle para acumular la suma de los términos. Como el valor de `sum` se actualiza en cada iteración y se utiliza en la siguiente iteración, no se puede declarar como privada, ya que cada hilo tendría su propia copia y los resultados finales no serían correctos. En cambio, se declara como compartida, lo que permite que todos los hilos accedan a la misma variable y actualicen su valor de manera segura. Se puede usar la cláusula **reduction** para realizar una operación de reducción en la variable compartida al final de cada iteración del bucle.
- **step**: La variable `step` es una constante global y no se actualiza en el bucle, pero si se utiliza en el cálculo de `pi`, por lo que puede ser una variable compartida.

2. ¿Qué realiza la función `omp_get_wtime()`?

Según [7], la función `omp_get_wtime()` devuelve el tiempo actual del sistema en segundos, y se utiliza para medir el tiempo de ejecución de una sección específica del

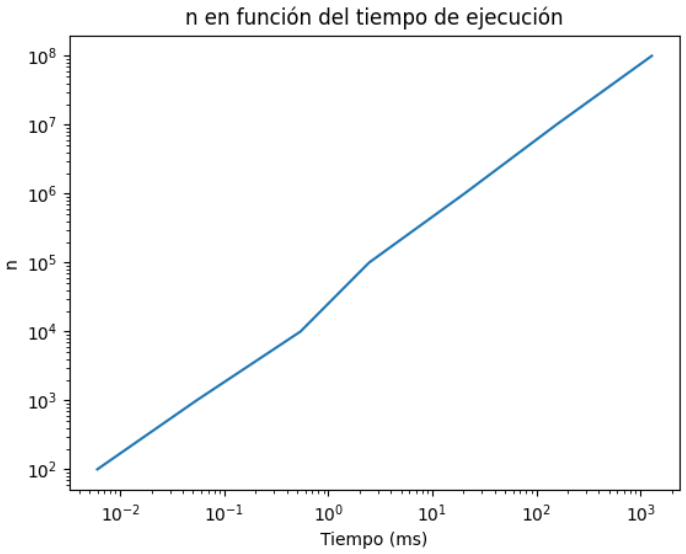
código. En el código dado, la función se utiliza para medir el tiempo total que tarda el bucle `for` en completarse y calcular el valor de `pi`.

Al guardar el tiempo inicial en la variable `start_time` y restarle al tiempo final obtenido después del bucle, se calcula el tiempo total de ejecución del bucle y se muestra al finalizar la ejecución. Esto es útil para comparar el rendimiento de diferentes implementaciones o configuraciones de paralelización en OpenMP.

3. Compile haciendo uso de OpenMP y ejecute el código modificando el parámetro de número de *steps*.
4. Mediante un gráfico de tiempo vs *steps*, pruebe 6 diferentes valores de *steps*. Explique brevemente el comportamiento ocurrido.

Muestras:

1. pi with 100 steps is 3.141601 in 0.000006 seconds
2. pi with 1000 steps is 3.141593 in 0.000054 seconds
3. pi with 10000 steps is 3.141593 in 0.000540 seconds
4. pi with 100000 steps is 3.141593 in 0.002466 seconds
5. pi with 1000000 steps is 3.141593 in 0.020307 seconds
6. pi with 10000000 steps is 3.141593 in 0.155542 seconds
7. pi with 100000000 steps is 3.141593 in 1.290154 seconds



El tiempo de ejecución depende de la cantidad de cálculos realizados en cada iteración del bucle `for`. A medida que se aumenta la cantidad de subintervalos, se realizan más cálculos, lo que aumenta el tiempo de ejecución. El aumento no es

exactamente lineal, pero sigue una escala logarítmica, lo que significa que el tiempo de ejecución aumenta aproximadamente en un orden de magnitud cada vez que se multiplica `num_steps` por diez. Además de que aumenta la precisión en el cálculo.

El código fuente `pi_loop.c` presenta el mismo cálculo, pero utilizando regiones paralelas. Para este segundo código debe realizar lo siguiente:

1. Explique cuál es el fin de los diferentes *pragmas* que se encuentran?

- **#pragma omp parallel:** Se utiliza para crear un equipo de hilos y la región paralela. Cada hilo en el equipo ejecuta el mismo código. Como se observa, dentro de esta directiva, se pueden utilizar otras directivas para controlar el comportamiento del equipo de hilos.
- **#pragma omp single:** Se utiliza para garantizar que solo se ejecute una vez la impresión del número de hilos que se están ejecutando en el equipo.
- **#pragma omp for:** Esta directiva se utiliza para dividir un bucle en varias iteraciones y distribuir las iteraciones a diferentes hilos para su ejecución. Cada hilo ejecuta una parte diferente del bucle. En este caso se emplea para paralelizar el bucle que calcula la suma de la serie para estimar el valor de Pi.
- **#reduction(+:sum):** Se utiliza dentro de la cláusula `for` para garantizar que la actualización de `sum` sea segura, Lo que ocurre aquí es que cada hilo mantiene una copia privada de la variable compartida y actualiza esta copia local en su sección de código. Al final de la sección paralela, todas las copias locales se combinan en una única variable global utilizando una operación específica de reducción (en este caso la suma). El resultado final se almacena en una variable privada.
- **private(x):** Se utiliza dentro de la cláusula `for` para declarar la variable `x` como privada para cada hilo. Esto significa que cada hilo tendrá su propia copia local de la variable `x`, en lugar de compartir una variable global.

2. ¿Qué realiza la función `omp_get_num_threads()`?

Según [6], es una función de la API de OpenMP que se utiliza para obtener el número de hilos en ejecución en un equipo en particular. En el código dado, se llama dentro de `pragma omp single` para imprimir el número de hilos en ejecución en ese momento. Por lo que, la llamada a `omp_get_num_threads()` se realiza solo una vez y se imprime el número de hilos que se utilizarán para la próxima sección paralela `omp for`.

3. En la línea 41, el ciclo se realiza 4 veces. Realice un cambio en el código fuente para que el ciclo se repita el doble de la cantidad de procesadores disponibles para el programa. Incluya un *screenshot* con el cambio.

```
num_threads = 1
pi is 3.141593 in 1.318327 seconds and 1 threads
num_threads = 2
pi is 3.141593 in 0.637734 seconds and 2 threads
num_threads = 3
pi is 3.141593 in 0.425955 seconds and 3 threads
num_threads = 4
pi is 3.141593 in 0.319474 seconds and 4 threads
num_threads = 5
pi is 3.141593 in 0.259087 seconds and 5 threads
num_threads = 6
pi is 3.141593 in 0.242976 seconds and 6 threads
num_threads = 7
pi is 3.141593 in 0.222826 seconds and 7 threads
num_threads = 8
pi is 3.141593 in 0.214885 seconds and 8 threads
```

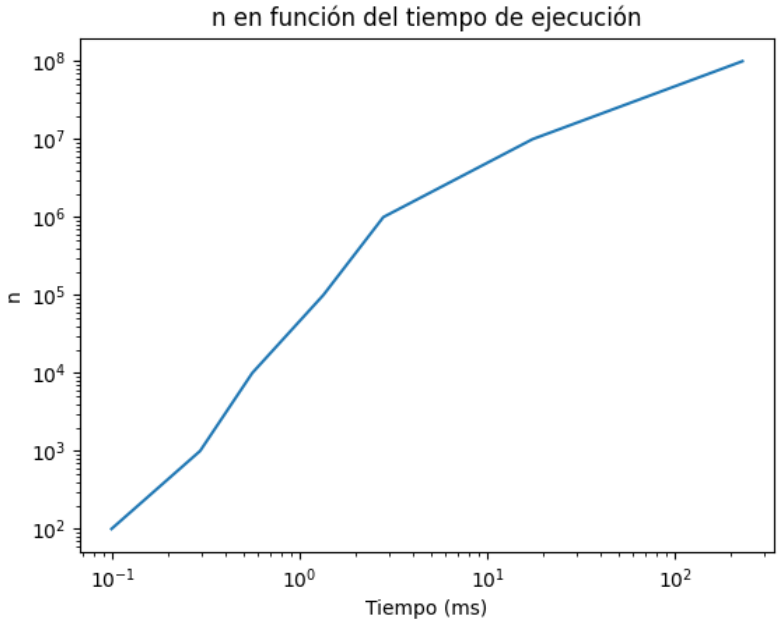
```
num_threads = 9
pi is 3.141593 in 0.246477 seconds and 9 threads
num_threads = 10
pi is 3.141593 in 0.224686 seconds and 10 threads
num_threads = 11
pi is 3.141593 in 0.216408 seconds and 11 threads
num_threads = 12
pi is 3.141593 in 0.223191 seconds and 12 threads
num_threads = 13
pi is 3.141593 in 0.215532 seconds and 13 threads
num_threads = 14
pi is 3.141593 in 0.216172 seconds and 14 threads
num_threads = 15
pi is 3.141593 in 0.209225 seconds and 15 threads
num_threads = 16
pi is 3.141593 in 0.210203 seconds and 16 threads
```

4. Compile haciendo uso de OpenMP y ejecute el código modificando el parámetro de número de *steps*.

5. Mediante un gráfico de tiempo vs *steps*, pruebe 6 diferentes valores de *steps*. Explique brevemente el comportamiento ocurrido.

Muestras:

1. pi with 100 steps is 3.141601 in 0.000099 seconds and 16 threads
2. pi with 1000 steps is 3.141593 in 0.000294 seconds and 16 threads
3. pi with 10000 steps is 3.141593 in 0.000558 seconds and 16 threads
4. pi with 100000 steps is 3.141593 in 0.001336 seconds and 16 threads
5. pi with 1000000 steps is 3.141593 in 0.002792 seconds and 16 threads
6. pi with 10000000 steps is 3.141593 in 0.017465 seconds and 16 threads
7. pi with 100000000 steps is 3.141593 in 0.229951 seconds and 16 threads



A medida que aumenta el número de subintervalos, el tiempo de ejecución aumenta de manera significativa. En general, el tiempo de ejecución aumenta proporcionalmente al número de subintervalos. Lo cual se debe a que el cálculo de cada subintervalo es independiente del cálculo de los demás subintervalos, lo que permite un procesamiento en paralelo eficiente utilizando OpenMP.

Se observa que el tiempo de ejecución aumenta a medida que se aumenta la precisión de la aproximación (al utilizar mayor número de subintervalos). No obstante, el aumento en el tiempo de ejecución no es lineal debido a la sobrecarga de administración de subprocesos, en algún punto se limita.

En definitiva, el comportamiento esperado de los resultados es que el tiempo de ejecución aumente proporcionalmente con el número de pasos y disminuya a medida que se aumenta el número de hilos utilizados. Sin embargo, hay una sobrecarga asociada con la paralelización que puede afectar el tiempo de ejecución y limita el beneficio del paralelismo.

6. Compare los resultados con el ejercicio anterior.

La comparación de los resultados en serie y en paralelo muestra que el tiempo de ejecución para el mismo valor de num_steps es significativamente más corto en la versión en paralelo. Esto se debe a que el uso de múltiples hilos permite que los cálculos se realicen en paralelo, lo que reduce el tiempo total de ejecución.

También se observa que a medida que aumenta el valor de num_steps, los tiempos de ejecución de las versiones serial y paralela también aumentan, pero la versión paralela sigue siendo significativamente más rápida.

4. Ejercicios prácticos

1. Realice un programa en C que aplique la operación SAXPY de manera serial y paralela (OpenMP). Compare el tiempo de ejecución de ambos programas para al menos tres tamaños diferentes de vectores.

Implementación de la operación SAXPY fue tomada de la sección 2.2.2 de [8].

```
jey@jey:~/Documents/Arquitectura2/Taller2$ gcc saxpy.c -o saxpy -fopenmp
jey@jey:~/Documents/Arquitectura2/Taller2$ ./saxpy
Serial   for n=10000            0.000026 seconds
Parallel for n=10000            0.000807 seconds

Serial   for n=100000           0.000374 seconds
Parallel for n=100000           0.000301 seconds

Serial   for n=1000000          0.003781 seconds
Parallel for n=1000000          0.003930 seconds

Serial   for n=10000000         0.033496 seconds
Parallel for n=10000000         0.012629 seconds

Serial   for n=100000000        0.260865 seconds
Parallel for n=100000000        0.090536 seconds
```

Se observa que el tiempo de ejecución serial, tiene un comportamiento lineal respecto al número de iteraciones, ahora bien, para el tiempo paralelo, se nota un decremento del tiempo de ejecución alrededor del `n = 100000`, lo cual indica que alrededor de este valor hay un rendimiento óptimo de los recursos del sistema, o bien la administración de los hilos no se sobrecarga y por ende hay un tiempo de ejecución excelente. Al aumentar o disminuir las iteraciones, tomando este punto como referencia, se ve cómo el tiempo empieza a aumentar.

Código

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

void initialize_vectors(float *x, float *y, int n) {
    for (int i = 0; i < n; i++) {
        x[i] = (float) rand() / RAND_MAX;
        y[i] = (float) rand() / RAND_MAX;
    }
}

void run_saxpy_serial(float a, float *x, float *y, int n) {
    double start_time = omp_get_wtime();
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
    double run_time = omp_get_wtime() - start_time;
    printf("Serial   for n=%d     \t%f seconds\n", n, run_time);
}

void run_saxpy_parallel(float a, float *x, float *y, int n) {
    double start_time = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        y[i] = a * x[i] + y[i];
    }
    double run_time = omp_get_wtime() - start_time;
    printf("Parallel for n=%d     \t%f seconds\n\n", n, run_time);
}

int main()
{
    int n1 = 10000;
    int n2 = 100000;
    int n3 = 1000000;

    float a = 2.0;
    float *x, *y;

    // Allocate memory for vectors
    x = (float *) malloc(n3 * sizeof(float));
    y = (float *) malloc(n3 * sizeof(float));

    // Initialize vectors with random values
    initialize_vectors(x, y, n3);

    // Run saxpy for n1
    run_saxpy_serial(a, x, y, n1);
    run_saxpy_parallel(a, x, y, n1);

    // Run saxpy for n2
    run_saxpy_serial(a, x, y, n2);
    run_saxpy_parallel(a, x, y, n2);

    // Run saxpy for n3
    run_saxpy_serial(a, x, y, n3);
    run_saxpy_parallel(a, x, y, n3);

    // Free memory
    free(x);
    free(y);

    return 0;
}
```

2. Realice un programa en C utilizando OpenMP para calcular el valor de la constante e. Compare los tiempos y qué tan aproximado al valor real para 6 valores distintos de n.

Para calcular el valor de la constante e, podemos utilizar la siguiente serie:

$$e = 1 + 1/1! + 1/2! + 1/3! + \ldots$$

```
jey@jey:~/Documents/Arquitectura2/Taller2$ gcc euler.c -o euler -fopenmp
jey@jey:~/Documents/Arquitectura2/Taller2$ ./euler
n               tiempo          euler                   presición
valor real                      2.7182818284590451
--------------------------------------------------------------------------------
n=50
serial:         0.000007        2.7182818284590455      0.0000000000000163
paralelo:       0.097013        3.0982852781881434      13.9795456729560961
--------------------------------------------------------------------------------
n=100
serial:         0.000012        2.7182818284590455      0.0000000000000163
paralelo:       0.007261        2.9181829271518747      7.3539504476675486
--------------------------------------------------------------------------------
n=500
serial:         0.000086        2.7182818284590455      0.0000000000000163
paralelo:       0.001451        2.7594733022232436      1.5153496349401478
--------------------------------------------------------------------------------
n=1000
serial:         0.000035        2.7182818284590455      0.0000000000000163
paralelo:       0.001600        2.7390240791754166      0.7630647602176690
--------------------------------------------------------------------------------
n=5000
serial:         0.000120        2.7182818284590455      0.0000000000000163
paralelo:       0.000055        2.7224303950073510      0.1526172343453301
--------------------------------------------------------------------------------
n=10000
serial:         0.000239        2.7182818284590455      0.0000000000000163
paralelo:       0.058459        2.7203561135628531      0.0763086844819144
--------------------------------------------------------------------------------
```

Se observa que la precisión de la ejecución serial permanece igual para todas las ejecuciones, mientras que la precisión paralela es más exacta, la cantidad de error disminuye a medida que aumenta el `n`. Respecto a los tiempos, se nota que para los valores cercanos al `n=5000`, los resultados de paralelización son buenos, los valores por debajo y por arriba de esta marca, muestran que existen limitaciones por la saturación. Y como se espera para el tiempo de la ejecución serial, va aumentando a medida que se incrementan las iteraciones.

Código

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

const double euler_real = exp(1);

// Calcula la suma de los términos de la serie de Euler hasta el n-ésimo término en serie (modo serial)
double euler_serial(int n) {
    double euler_sum = 1.0, term = 1.0, fact = 1.0;
    double error;

    for (int i = 1; i <= n; i++) {
        fact *= i;
        term = 1.0 / fact;
        euler_sum += term;
    }
    return euler_sum;

}

// Calcula la suma de los términos de la serie de Euler hasta el n-ésimo término en serie (modo paralelo)
double euler_parallel(int n) {
    double euler_sum = 1.0, term = 1.0, fact = 1.0;
    double error;

    #pragma omp parallel for reduction(*:fact) reduction(+:euler_sum)
    for (int i = 1; i <= n; i++) {
        fact *= i;
        term = 1.0 / fact;
        euler_sum += term;
    }
    return euler_sum;
}


int main() {
    int n[] = {50, 100, 500, 1000, 5000, 10000};
    double e_serial, e_parallel, time_serial, time_parallel;

    printf("n\t\ttiempo\t\teuler\t\t\tpresición\n");
    printf("\033[0;31mvalor real\t\t\t%.16f\033[0m\n", euler_real);
    printf("--------------------------------------------------------------------------------");

    for (int i = 0; i < 6; i++) {
        clock_t start, end;
        double e1,e2,pr1,pr2;

        start = clock();
        e1 = euler_serial(n[i]);
        time_serial = ((double) (clock() - start)) / CLOCKS_PER_SEC;

        start = clock();
        e2 = euler_parallel(n[i]);
        time_parallel = ((double) (clock() - start)) / CLOCKS_PER_SEC;

        pr1 = fabs((e1 - euler_real) / euler_real) * 100;
        pr2 = fabs((e2 - euler_real) / euler_real) * 100;

        printf("n=%d\n", n[i]);
        printf("serial:\t\t%f\t%.16f\t%.16f\n", time_serial, e1, pr1);
        printf("paralelo:\t%f\t%.16f\t%.16f\n", time_parallel, e2, pr2);
        printf("--------------------------------------------------------------------------------");
    }

    return 0;
}
```

References

[1] "What is OpenMP?," *www.tutorialspoint.com*. https://www.tutorialspoint.com/what-is-openmp

[2] "Intro to Parallel Programming with OpenMP," Bowdoin.edu, 2023. https://tildesites.bowdoin.edu/%7Eltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html#:%7E:text=When%20run%2C%20an%20OpenMP%20program.

[3] "OpenMP Tutorial - Critical, Atomic and Reduction," *Ive Moved*, Oct. 23, 2011. https://michaellindon.github.io/lindonslog/programming/openmp/openmp-tutorial-critical-atomic-and-reduction/index.html.

[4] "Reduction Clauses and Directives," *www.openmp.org*. https://www.openmp.org/spec-html/5.0/openmpsu107.html.

[5] "The Flush Operation," *www.openmp.org*. https://www.openmp.org/spec-html/5.0/openmpsu12.html.

[6] "omp_get_num_threads," www.openmp.org. https://www.openmp.org/spec-html/5.0/openmpsu111.html.

[7] "omp_get_wtime," *www.openmp.org*. https://www.openmp.org/spec-html/5.0/openmpsu160.html.

[8] M. J. Quinn, "Parallel Programming in C with MPI and OpenMP," McGraw-Hill Education, 2004, ISBN: 978-0071232654.