Tecnológico de Costa Rica
Área académica de Ingeniería En Computadores

Taller 2 - OpenMP

Curso
CE 4302 - Arquitectura de Computadores II

Realizado por
Luis López Salas - 2015088115

Profesor
Luis Barboza Artavia

Abril 2021

1. ¿En qué consiste OpenMP?
OpenMP es un API (application programming interface/interfaz para la programación de aplicaciones) que permite multiprocesamiento con memoria compartida. Está disponible en varias plataformas. Se puede utilizar en C/C++ o Fortran. Es portable y escalable, lo que simplifica el desarrollo de aplicaciones que necesiten este.

2. ¿Cómo se define la cantidad de hilos en OpenMP?
La variable de ambiente OMP_NUM_THREADS se usa para definir la cantidad de hilos. Este se utiliza a la hora de ejecutar el código una vez compilado. Esto se debe hacer por línea de comandos: "*OMP_NUM_THREADS=12 ./omp_code.exe"* para hacerlo únicamente una vez y "*export OMP_NUM_THREADS=24"* para cambiar la variable permanentemente. Este se define por dentro por medio de la variable interna (ICV) *nthreads-var.*

3. ¿Cómo se crea una región paralela en OpenMP?
Para crear una región paralela ésta se debe hacer dentro del código. Se debe declarar primero "#pragma omp parallel" y seguidamente el código que se desea ejecutar de manera paralela. De manera:

```
#pragma omp parallel
{
        //Código parralelo
        printf("Hello world\n");
}
```

4. ¿Cómo se compila un código fuente C para utilizar OpenMP y qué encabezado debe incluirse?
Para compilar un código que utilice OpenMP en C se debe de poner las banderas e incluir los parametros correctos. Algunas banderas que se usan se incluyen en la siguiente tabla la bandera depende del compilador que se usa:

Compiler	Flag
GNU	-fopenmp
Intel	-qopenmp
Clang	-fopenmp
Oracle	-xopenmp
NAG Fortran	-openmp

Debido a que usualmente se utiliza Linux se hará un ejemplo con el compilador GNU. Ej:

```
gcc -fopenmp omp_code.c -o omp_code.exe
```

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.

Un programa puede utilizar como máximo la cantidad de procesadores que tiene una máquina. Para poder revisar[1] la cantidad de procesadores que tiene una máquina se proveen los siguientes comandos:
$ lscpu
$ lscpu | egrep 'Model name|Socket|Thread|NUMA|CPU\(s\)' (Recomendado)
$ lscpu -p
El cual provee un resultado de la forma:

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos$ lscpu | egrep 'Model name|Socket|Thread|NUMA|CPU\(s\)'
CPU(s):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Socket(s):                       1
NUMA node(s):                    1
Model name:                      AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
NUMA node0 CPU(s):               0-7
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos$
```

6. ¿Cómo se definen las variables privadas en OpenMP?¿Por qué son importantes?
Las variables privadas (por lo menos en C) se definien por medio de la palabra reservada “private”. Esta se coloca en la misma línea en que se escribe la declaración #pragma omp parallel de la siguiente manera:
#pragma omp parallel private(id, np)
La importancía de tener variables privadas es que son independientes. Es decir, solo las puede editar un thread (hilo) a cual pertenecen. Lo cual puede ser importante si cada thread está realizado su propio cálculo y necesita sus propios parametros unicos o su propias salidas unicas. Si no se declara una variable como privada, es compartida. Por lo que cualquier hilo la puede editar, lo cual puede dar errores a la hora de ejecución. Las variables definidas dentro de una región paralela son por defecto privadas.

7. ¿Cómo se definen las variables compartidas en OpenMP?¿Cómo se deben actualizar valores de variables compartidas?
Para variables compartidas es un proceso similar al de variables privadas. Únicamente que estas llevan el nombre de “shared” en vez de “private”[2]. Similarmente a las variables privadas se consideran que las variables son compartidas si estas se definen fuera de la región de paralelización. A menos que se utilice la palabra “Default” que puede cambiar esto.
#pragma omp parallel for private(temp) shared(n,a,b,c)
Si se va a actualizar una variable compartida se considerabuena práctica utilizar la instrucción **#pragma omp critical(name)**. Este es un tipo de mutex, que permite que sólo un thread tenga acceso a esta variable. El nombre (“name”) se puede utilizar cuando son varios mutex. De forma que estos no “choquen”.

8. ¿Para qué sirve flush en OpenMP?
Debido a que no es necesario que una variable siempre esté actualizada a lo que está en memoria para su funcionamiento. OpenMP permite que lasvariables funcionen sin estar actualizadas siempre. A esto se le conoce como vista, la vista del hilo se le llama la vista privada temporanea del hilo. [4]El flush es un set de instrucciones que permiten al programador manejar cómo es que se actualizan las variables. Por ejemplo, para forzar que un hilo siempre tenga la variable actualizada se utiliza el strong-flush. Las variables que

tengan flush se llaman flush-set o set de flush. Hay más funciones (e.g. release flush) pero el strong flush es el más común.

9. ¿Cuál es el propósito del *pragma omp single*? ¿En cuáles casos debe usarse?
El single construct (*pragma omp single*)[5] indica que el bloque asociado solamente se necesita ejecutar una vez por un único hilo. Este no es necesariamente el hilo principal. Los demás hilos deben de esperar a que este termine antes de continuar (a menos que se utilice “nowait”).

Se debe utilizar al principio de un programa donde se estpan creando todos los hilos que se van a utilizar. Esto asegura que solo se creen la cantidad necesaria y tengan los datos correctos. También permite dividir los hilos de manera que cada uno realice una tarea separada.

10. ¿Cuáles son tres instrucciones para la sincronización? Realice una comparación entre ellas donde incluya en cuáles casos se utiliza cada una.
Tres instrucciones[6] utilizadas comúnmente son:

- **Barrier:** Un barrier o barrera define un punto en el c[odigo donde todos los hilos deben esperar. No se puede continuar hasta que todos los hilos alcancen a este punto. Esto garantiza que se cumpla cierto cálculo antes de continuar.
- **Mutex:** Un mutex se diferencia en que solo permite que un hiilo ejecute la región de código a la vez. A esta porción del código se le designa como crítica. OpenMP tiene varios mecanismos para lograr un mutex.
- **Lock:** Aparte de Mutex OpenMP tambien tiene la opción de lock/candado. Es similar al mutex, en que se asegura que solo una parte del código se ejecute solamente por un hilo. Sin embargo, el lock se refiere a datos más que a código. Lo que permite la ejecución de código, sin embargo, solo un hilo tiene acceso al dato.

11. ¿Cuál es el propósito de *reduction* y cómo se define?
La “clausula” de reducción[4] son un atributo que se utiliza para compartir datos. Esto permite que se realicen cálculos en paralelo utilizando los mismos datos. Hay dos tipos de participación y alcance. El alcance se define una región del código que es definida como recuction. En cambio, la de participación define cuales son los participantes que tienen acceso a esta región. Se puede pensar como una variable global que solo se modifica cuando termina la ejecución de todos los hilos.
#pragma omp parallel for reduction(+:sum)

pi.c:

3.1 Identifique cuáles secciones se pueden paralelizar y las variables pueden ser privadas o compartidas.

Las secciones que se pueden paralelizar son cada son las instrucciones que se encuentran dentro del for. Estas son las líneas 32 y 33. Debido a que no dependen de alguna otra región y se tienen que realizar multiples veces. No tendría sentido paralelizar el print, debido a que solo se realiza una vez.

Las variables **privadas** deben ser la variable “x” y la variable “i”. Debido a que son únicas su propio hilo. Cada una va a tener estas variables pero sus valores van a ser independientes. De otra forma, se estaría calculando el mismo valor y no tendría sentido la paralelización.

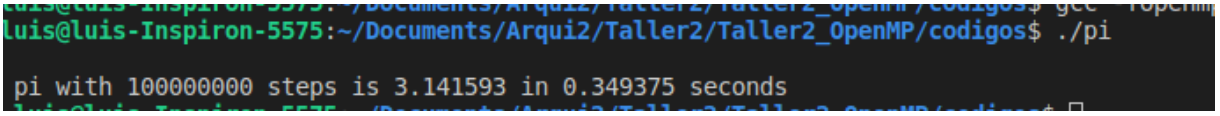
Las variables **compartidas** son las variables “step” y “sum”. La variable step es la misma para todos los hilos. Esta únicamente se lee y no se actualiza. Por lo que se puede leer por cualquiera sin problema. En cambio, se está acutalizando constantemente la variable sum y debe ser la misma para todos los hilos. Es el resultado, se podría sumar todas estas al final o sumar cada vez que termine un hilo. Dependiendo de como se diseñe puede ser necesario un lock o un reduction.

3.2 ompgetwtime()

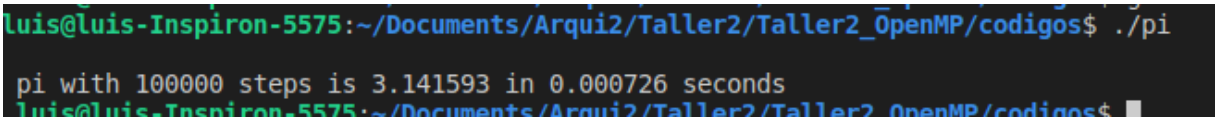
Devuelve la cantidad de tiempo que ha transcurrido desde el tiempo inicial del sistema operativo. Se puede utilizar esta función dos veces para calcular la diferencia que se tiene de un tiempo a otro. Es útil para verificar cuento fue el tiempo que se tardo en realizar una tarea.

3.3 Compilación y modificación del código.

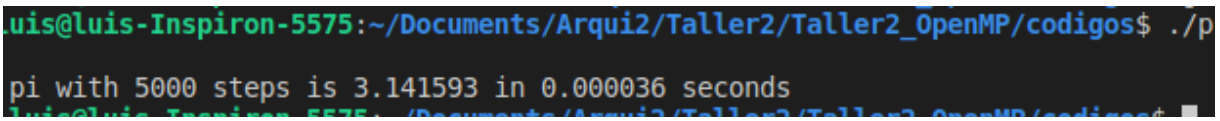
Con 100000000 steps:



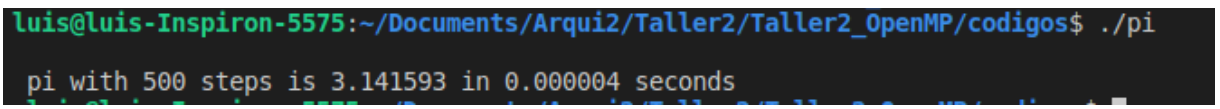
Con 100000



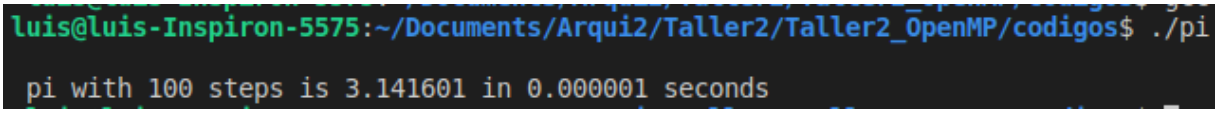
Con 5000



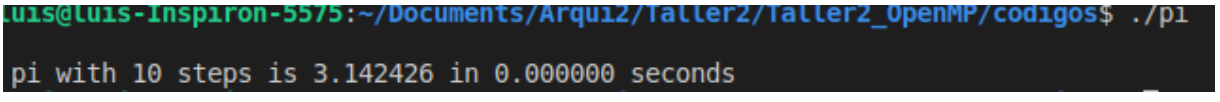
500



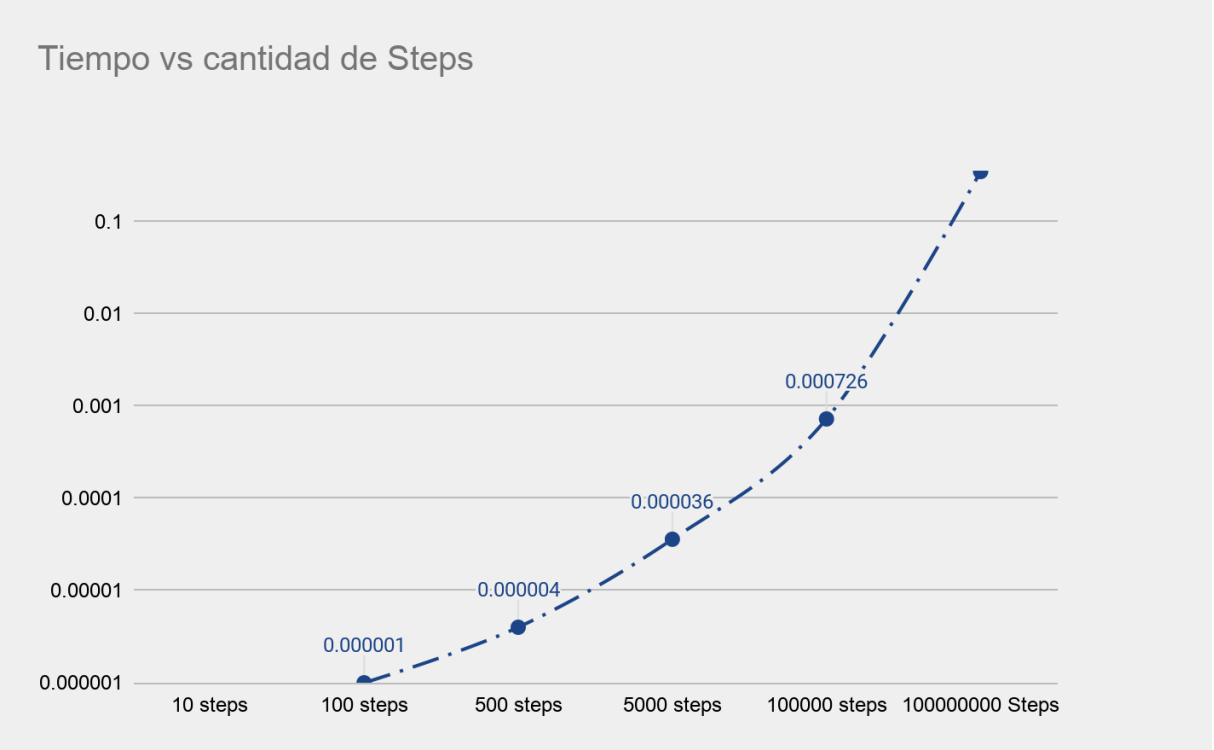
100



10



3.4 Gráfico:



Como se puede apreciar claramente en el grafico, entre más cantidad de steps o “pasos” se hagan más va a tardar el programa. Esto se debe a que debido a que se estan haciendo más iteraciones su tiempo de ejecución aumenta. Sin embargo, tiene la ventaja de que el resultado es más exacto. El impacto no se sufre directamente en la variable num_steps sino en la variable steps y por ende la x. Lo que termina impactando a sum, la respuesta. Lo que hace num_steps es incrementar poco a poco el valor de x, haciendo que la integral se calcule de manera más exacta, entre mayor el paso, menor la precisión.

pi_loop.c:

3.1Pragmas:

#pragma omp parallel:

Indica la región paralelizable del código. Toda región que esté dentro de estas llaves se va a ejecutar en sus propio hilo.

#pragma omp single:

Indica una región dentro de la región paralelizable que se debe ejecutar una única vez. No importa por cuál hilo, pero se ejecuta una vez. En este caso se utiliza para mostrar el número de threads que están corriendo en la misma región. No sería muy comodo para el usuario ver el resultado 8 veces seguidas.

#pragma omp for reduction(+:sum):

Permite que se comparta de manera segura la variable sum, por medio de la acción de “+”. Esto para que no se tenga un dato erroneo al final, debido a que se ignoraron datos o incluso se repitieron.

3.2 **ompgetnumthreads()**

Esta función retorna la cantidad de hilos que actualmente se tiene dentro de la misma región paralela. En este caso se utiliza para hacer un print con el número de threads que están corriendo.

3.3

```
*/
#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;
      int procnum = 8;
     for (i=1;i<=procnum*2;i++){
          sum = 0.0;
          omp_set_num_threads(i);
      start_time = omp_get_wtime();
#pragma omp parallel
{
#pragma omp single
      printf(" num_threads = %d",omp_get_num_threads());

#pragma omp for reduction(+:sum)
      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 is %f in %f seconds and %d threads\n",pi,run_time,i);
}
}
```

3.4

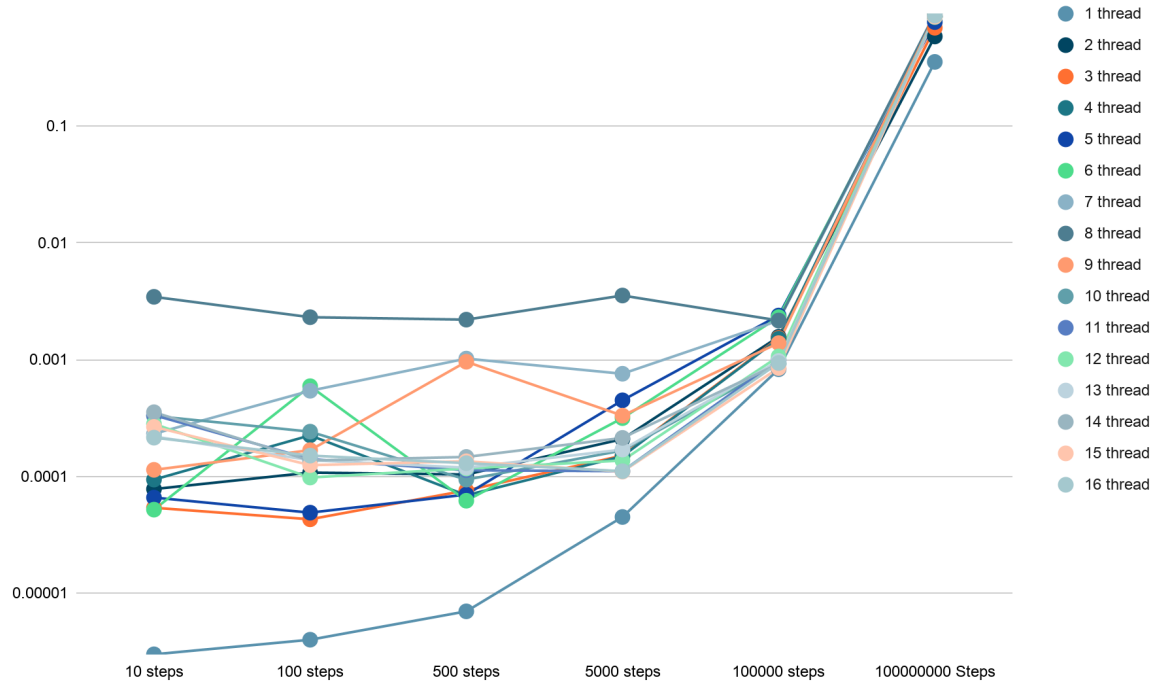
Se probaron los mismos valores, sin embargo no se va a mostrar todos debido a que los resultados son muy largos. Los valores son:

- 100000000
- 100000
- 5000
- 500
- 100
- 10

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos$ gcc -fopenmp pi_loop.c -o pi_loop
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos$ ./pi_loop
 num_threads = 1
 pi is 3.141593 in 0.361281 seconds and 1 threads, num_steps = 100000000
 num_threads = 2
 pi is 3.141904 in 0.664070 seconds and 2 threads, num_steps = 100000000
 num_threads = 3
 pi is 3.141620 in 0.690816 seconds and 3 threads, num_steps = 100000000
 num_threads = 4
 pi is 3.141619 in 0.751594 seconds and 4 threads, num_steps = 100000000
 num_threads = 5
 pi is 3.145031 in 0.720200 seconds and 5 threads, num_steps = 100000000
 num_threads = 6
 pi is 3.142892 in 0.828983 seconds and 6 threads, num_steps = 100000000
 num_threads = 7
 pi is 3.151807 in 0.837221 seconds and 7 threads, num_steps = 100000000
 num_threads = 8
 pi is 3.151789 in 0.825655 seconds and 8 threads, num_steps = 100000000
 num_threads = 9
 pi is 3.147440 in 0.820332 seconds and 9 threads, num_steps = 100000000
 num_threads = 10
 pi is 3.152373 in 0.853346 seconds and 10 threads, num_steps = 100000000
 num_threads = 11
 pi is 3.150352 in 0.857608 seconds and 11 threads, num_steps = 100000000
 num_threads = 12
 pi is 3.149449 in 0.869722 seconds and 12 threads, num_steps = 100000000
 num_threads = 13
 pi is 3.151820 in 0.885530 seconds and 13 threads, num_steps = 100000000
 num_threads = 14
 pi is 3.149278 in 0.844481 seconds and 14 threads, num_steps = 100000000
 num_threads = 15
 pi is 3.147255 in 0.839004 seconds and 15 threads, num_steps = 100000000
 num_threads = 16
 pi is 3.151012 in 0.869054 seconds and 16 threads, num_steps = 100000000
```

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos$ gcc -fopenmp pi_loop.c -o pi_loop
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos$ ./pi_loop
 num_threads = 1
 pi is 3.142426 in 0.000023 seconds and 1 threads, num_steps = 10
 num_threads = 2
 pi is 3.142426 in 0.000085 seconds and 2 threads, num_steps = 10
 num_threads = 3
 pi is 3.142426 in 0.000082 seconds and 3 threads, num_steps = 10
 num_threads = 4
 pi is 3.142426 in 0.000088 seconds and 4 threads, num_steps = 10
 num_threads = 5
 pi is 3.142426 in 0.000052 seconds and 5 threads, num_steps = 10
 num_threads = 6
 pi is 3.142426 in 0.000056 seconds and 6 threads, num_steps = 10
 num_threads = 7
 pi is 3.142426 in 0.000796 seconds and 7 threads, num_steps = 10
 num_threads = 8
 pi is 3.142426 in 0.003125 seconds and 8 threads, num_steps = 10
 num_threads = 9
 pi is 3.142426 in 0.000379 seconds and 9 threads, num_steps = 10
 num_threads = 10
 pi is 3.142426 in 0.000174 seconds and 10 threads, num_steps = 10
 num_threads = 11
 pi is 3.142426 in 0.000178 seconds and 11 threads, num_steps = 10
 num_threads = 12
 pi is 3.142426 in 0.000267 seconds and 12 threads, num_steps = 10
 num_threads = 13
 pi is 3.142426 in 0.000191 seconds and 13 threads, num_steps = 10
 num_threads = 14
 pi is 3.142426 in 0.000190 seconds and 14 threads, num_steps = 10
 num_threads = 15
 pi is 3.142426 in 0.000172 seconds and 15 threads, num_steps = 10
 num_threads = 16
 pi is 3.142426 in 0.000172 seconds and 16 threads, num_steps = 10
```

3.5



En el grafico se muestran 6 corridas diferentes donde se cambian el número máximo de steps que se toman. Es similar a la grafica anterior únicamente que este toma en cuenta hilos.

Se observa que el mejor rendimiento se da con un hilo. Puede sonar ilógico. Si se estan usando más hilos debería de ser más rápido. Sin embargo, es dependiente del tipo de tarea que se utilice así como el hardware. Puede ser un artefacto único a la máquina en la que se probó. No obstantemente, hay otra razón, puede ser que la aplicación sea muy simple y las instrucciones que se están ejecutando no necesiten tanto poder computacional. Lo que hace que el *“overhead” tenga más peso que el código a ejecutar.

*Overhead se refiere a cómo es que el computador debe tratar los hilos (e.g. la creación de estos o como hacer para que compartan datos). Sea a nivel de hardware o a nivel de sistema operativo.

3.6 Comparación hilos vs sin hilos

En esta aplicación, según los resultados, tiene una mejor ejecución sin hilos. Esto puede ser por varios factores. Sin embargo, se considera que lo más importante es cómo se debe organizar el sistema operativo/hardware para correr los hilos. Los hilos son herramientas extremadamente útiles, pero se deben de aplicar de forma corr ecta. En un caso donde la computación fuera más pesada se podría observar el comportamiento opuesto.

4.1 Saxpy

Serial

```
hereluis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp saxpy.c -o saxpy
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./saxpy
RunTime = 0.000795, with array size of = 100000
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp saxpy.c -o saxpy
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./saxpy
RunTime = 0.000009, with array size of = 1000
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp saxpy.c -o saxpy
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./saxpy
RunTime = 0.000001, with array size of = 50
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$
```

Paralelo

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp saxpy_loop.c -o saxpy_loop
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./saxpy_loop
threads = 1 RunTime = 0.001078, with array size of = 100000
threads = 2 RunTime = 0.000991, with array size of = 100000
threads = 3 RunTime = 0.001101, with array size of = 100000
threads = 4 RunTime = 0.001845, with array size of = 100000
threads = 5 RunTime = 0.001210, with array size of = 100000
threads = 6 RunTime = 0.002097, with array size of = 100000
threads = 7 RunTime = 0.002280, with array size of = 100000
threads = 8 RunTime = 0.004834, with array size of = 100000
threads = 9 RunTime = 0.001801, with array size of = 100000
threads = 10 RunTime = 0.001656, with array size of = 100000
threads = 11 RunTime = 0.001950, with array size of = 100000
threads = 12 RunTime = 0.001866, with array size of = 100000
threads = 13 RunTime = 0.001948, with array size of = 100000
threads = 14 RunTime = 0.002232, with array size of = 100000
threads = 15 RunTime = 0.002103, with array size of = 100000
threads = 16 RunTime = 0.002388, with array size of = 100000
```

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp saxpy_loop.c -o saxpy_loop
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./saxpy_loop
threads = 1 RunTime = 0.000032, with array size of = 1000
threads = 2 RunTime = 0.000084, with array size of = 1000
threads = 3 RunTime = 0.000074, with array size of = 1000
threads = 4 RunTime = 0.000081, with array size of = 1000
threads = 5 RunTime = 0.000078, with array size of = 1000
threads = 6 RunTime = 0.000401, with array size of = 1000
threads = 7 RunTime = 0.000709, with array size of = 1000
threads = 8 RunTime = 0.005438, with array size of = 1000
threads = 9 RunTime = 0.000156, with array size of = 1000
threads = 10 RunTime = 0.000119, with array size of = 1000
threads = 11 RunTime = 0.000116, with array size of = 1000
threads = 12 RunTime = 0.000112, with array size of = 1000
threads = 13 RunTime = 0.000575, with array size of = 1000
threads = 14 RunTime = 0.000216, with array size of = 1000
threads = 15 RunTime = 0.000291, with array size of = 1000
threads = 16 RunTime = 0.000215, with array size of = 1000
```

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp saxpy_loop.c -o saxpy_loop
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./saxpy_loop
threads = 1 RunTime = 0.000022, with array size of = 50
threads = 2 RunTime = 0.000068, with array size of = 50
threads = 3 RunTime = 0.000041, with array size of = 50
threads = 4 RunTime = 0.000072, with array size of = 50
threads = 5 RunTime = 0.000080, with array size of = 50
threads = 6 RunTime = 0.000056, with array size of = 50
threads = 7 RunTime = 0.001447, with array size of = 50
threads = 8 RunTime = 0.004370, with array size of = 50
threads = 9 RunTime = 0.000626, with array size of = 50
threads = 10 RunTime = 0.000250, with array size of = 50
threads = 11 RunTime = 0.000222, with array size of = 50
threads = 12 RunTime = 0.000295, with array size of = 50
threads = 13 RunTime = 0.000288, with array size of = 50
threads = 14 RunTime = 0.000178, with array size of = 50
threads = 15 RunTime = 0.000146, with array size of = 50
threads = 16 RunTime = 0.000162, with array size of = 50
```

Se puede observar que en este tambien se pierde mucho procesamiento en cuanto overhead. Sin embargo, se nota que en un número mayor de iteraciones posiblemente el sistema paralelo genere ventaja sobre el serial. Otra factor a notar es que el tiempo de ejecución del paralelo fluctua. Esto se puede dar debido a que sí hay un gra numero de hilos uno puede esperar más que otro.

4.2

Serial

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp const_e.c -o const_e
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./const_e
2.718282
RunTime = 0.000047, with steps = 60
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp const_e.c -o const_e
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./const_e
2.718282
RunTime = 0.000039, with steps = 30
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp const_e.c -o const_e
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./const_e
2.718282
RunTime = 0.000032, with steps = 10
```

Paralelo

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp const_e_loop.c -o const_e_loop
^[[A^[[Bluis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./const_e_loop
threads = 1 RunTime = 0.000039, with steps = 60
threads = 2 RunTime = 0.000096, with steps = 60
threads = 3 RunTime = 0.000066, with steps = 60
threads = 4 RunTime = 0.000687, with steps = 60
threads = 5 RunTime = 0.000064, with steps = 60
threads = 6 RunTime = 0.000072, with steps = 60
threads = 7 RunTime = 0.001000, with steps = 60
threads = 8 RunTime = 0.003576, with steps = 60
threads = 9 RunTime = 0.000695, with steps = 60
threads = 10 RunTime = 0.000131, with steps = 60
threads = 11 RunTime = 0.000475, with steps = 60
threads = 12 RunTime = 0.000107, with steps = 60
threads = 13 RunTime = 0.000426, with steps = 60
threads = 14 RunTime = 0.000289, with steps = 60
threads = 15 RunTime = 0.000294, with steps = 60
threads = 16 RunTime = 0.000277, with steps = 60
```

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp const_e_loop.c -o const_e_loop
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./const_e_loop
threads = 1 RunTime = 0.000022, with steps = 30
threads = 2 RunTime = 0.000114, with steps = 30
threads = 3 RunTime = 0.000095, with steps = 30
threads = 4 RunTime = 0.000078, with steps = 30
threads = 5 RunTime = 0.000060, with steps = 30
threads = 6 RunTime = 0.000289, with steps = 30
threads = 7 RunTime = 0.000506, with steps = 30
threads = 8 RunTime = 0.003119, with steps = 30
threads = 9 RunTime = 0.000183, with steps = 30
threads = 10 RunTime = 0.000261, with steps = 30
threads = 11 RunTime = 0.000198, with steps = 30
threads = 12 RunTime = 0.000193, with steps = 30
threads = 13 RunTime = 0.000167, with steps = 30
threads = 14 RunTime = 0.000140, with steps = 30
threads = 15 RunTime = 0.000152, with steps = 30
threads = 16 RunTime = 0.000153, with steps = 30
```

```
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ gcc -fopenmp const_e_loop.c -o const_e_loop
luis@luis-Inspiron-5575:~/Documents/Arqui2/Taller2/Taller2_OpenMP/codigos/Codes$ ./const_e_loop
threads = 1 RunTime = 0.000020, with steps = 10
threads = 2 RunTime = 0.000090, with steps = 10
threads = 3 RunTime = 0.000052, with steps = 10
threads = 4 RunTime = 0.000059, with steps = 10
threads = 5 RunTime = 0.000062, with steps = 10
threads = 6 RunTime = 0.000855, with steps = 10
threads = 7 RunTime = 0.001714, with steps = 10
threads = 8 RunTime = 0.003227, with steps = 10
threads = 9 RunTime = 0.000336, with steps = 10
threads = 10 RunTime = 0.000108, with steps = 10
threads = 11 RunTime = 0.000534, with steps = 10
threads = 12 RunTime = 0.000124, with steps = 10
threads = 13 RunTime = 0.000444, with steps = 10
threads = 14 RunTime = 0.000113, with steps = 10
threads = 15 RunTime = 0.000108, with steps = 10
threads = 16 RunTime = 0.005425, with steps = 10
```

Una vez más se presenta un comportamiento similar. Se propone las mismas razones previamente mencionadas.

Referencias :

[1]Gite, V. (2020, November 11). Check how many CPUs are there in Linux system. Retrieved April 18, 2021, from https://www.cyberciti.biz/faq/check-how-many-cpus-are-there-in-linux-system/

[2]Schwarz, S. (n.d.). Shared vs. private variables. Retrieved April 19, 2021, from https://www.dartmouth.edu/~rc/classes/intro_openmp/shared_and_private.html

[3]Barney, B. (2010). Introduction to OpenMP. Retrieved April 18, 2021, from https://people.cs.pitt.edu/~melhem/courses/xx45p/OpenMp.pdf

[4]OpenMP (n.d.). The flush operation. Retrieved April 19, 2021, from https://www.openmp.org/spec-html/5.0/openmpsu12.html

[5]OpenMP (n.d.). Single Construct. Retrieved April 19, 2021, from https://www.openmp.org/spec-html/5.0/openmpsu38.html#x60-1090002.8.2

[6]Eijkhout, V. (2011). Openmp topic: Synchronization. Retrieved April 19, 2021, from https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-sync.html