

1. Investigación	2
1.1 ¿En qué consiste OpenMP?	2
1.2 ¿Cómo se defines la cantidad de hilos en OpenMP?	2
1.3 ¿Cómo se crea una región paralela en OpenMP?	2
1.4 ¿Cómo se compila un código fuente en C para utilizar OpenMP?	3
1.5 ¿Cuál función me permite conocer el número de procesadores disponibles para el programa?	3
1.6 ¿Cómo se define una variable privada en openMP?	3
1.7 ¿Cómo se definen las variables compartidas en OpenMP?	4
1.8 ¿Para qué sirve flush en OpenMP	4
1.8.1 Strong Flush	4
1.8.2 Release Flush	4
1.8.3 Acquire Flush	4
1.9 ¿Cuál es el propósito de pragma omp simple?	5
1.10 ¿Cuáles son tres instrucciones para la sincronización?	5
1.10.1 Regiones críticas	5
1.10.2 Barreras	5
1.10.3 Regiones Ordenadas	6
1.11 ¿Cuál es el propósito de reduction y cómo se define?	6
2. Análisis	7
2.1a ¿Qué secciones se pueden paralelizar y cuáles variables pueden ser privadas y cuáles compartidas en pi.c?	7
2.2a ¿Qué función realiza omp_get_wtime()?	7
2.3a Compile y modifique el número de steps	8
2.4a Grafique el número de pasos contra el tiempo	8
2.1b Explique los pragmas en pi_loop.c	9
2.2b ¿Qué realiza la función omp_get_num_threads()?	9
2.3b Modifique el programa para que se ejecute con el doble de procesadores disponibles.	9
2.4b Ejecute el código modificando el parámetro de pasos	10
2.5b Grafique el número de pasos contra el tiempo. Explique el comportamiento ocurrido.	10
2.6b Compare los resultados con el ejercicio anterior	10
3. Ejercicios prácticos	11
3.1 SAXPY	11
3.2 Aproximación del número de Euler	11

1. Investigación

1.1 ¿En qué consiste OpenMP?

OpenMP es una especificación para un conjunto de directivas de compilación, rutinas de bibliotecas y variables de ambiente que pueden ser utilizadas para especificar paralelismo de alto nivel en programas escritos con Fortran, C y C++.

OpenMP es el estándar más utilizado en sistemas de multiprocesamiento simétrico (SMP), es relativamente pequeño, soporta paralelismo incremental y muchas investigaciones se llevan a cabo en OpenMP por lo que está al día con los últimos desarrollos de hardware.

[\[https://www.openmp.org/about/openmp-faq/#WhatIs\]](https://www.openmp.org/about/openmp-faq/#WhatIs)

1.2 ¿Cómo se define la cantidad de hilos en OpenMP?

Por medio de la variable de ambiente OMP_NUM_THREADS. En la terminal donde se va a ejecutar el programa se define la variable con:

```
set OMP_NUM_THREADS=<number of threads>
```

En algunas consolas puede que sea necesario exportar este valor. Esto se hace por medio de:

[\[https://software.intel.com/content/www/us/en/develop/documentation/mkl-windows-developer-guide/top/managing-performance-and-memory/improving-performance-with-threading/setting-the-number-of-threads-using-an-openmp-environment-variable.html\]](https://software.intel.com/content/www/us/en/develop/documentation/mkl-windows-developer-guide/top/managing-performance-and-memory/improving-performance-with-threading/setting-the-number-of-threads-using-an-openmp-environment-variable.html)

```
export OMP_NUM_THREADS=<number of threads to use>.
```

Para modificar esta variable desde el código se utiliza:

```
omp_set_num_threads(<number of threads>);
```

Esta variable define el límite superior que openMP va a usar pero no asegura que se utilice esa cantidad de hilos. Para forzar el número de threads se debe deshabilitar los grupos dinámicos además de modificar la variable de ambiente con:

[\[https://stackoverflow.com/questions/11095309/openmp-set-num-threads-is-not-working\]](https://stackoverflow.com/questions/11095309/openmp-set-num-threads-is-not-working)

```
omp_set_dynamic(0);
```

1.3 ¿Cómo se crea una región paralela en OpenMP?

En C y C++ la región está delimitada por un pragma previo a un bloque estructurado. Este bloque sería la región paralela.

```
#pragma omp parallel [clause[ [,] clause] ... ] new-line  
    Structured-block
```

Donde "clause" puede ser 1 de las siguientes opciones:

```
if([parallel :] scalar-logical-expression)
```

```

num_threads(scalar-integer-expression)
default(private | firstprivate | shared | none)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction([reduction-modifier ,] reduction-identifier : list)
proc_bind(master | close | spread)
allocate([allocator :] list)

```

Algunas restricciones a la construcción paralela en C son:

- Un programa que entra o sale de la región paralela no es conforme
- El programa debe ser independiente del orden en que se ejecuten las instrucciones en la región paralela.
- Como máximo una (1) sola cláusula if puede estar en la directiva
- Como máximo una (1) sola cláusula proc_bind puede estar en la directiva
- Como máximo una (1) sola cláusula num_threads puede estar en la directiva y la expresión se debe poder evaluar a un número entero positivo.

```

omp_set_num_threads(4);
sum = 0;
#pragma omp for reduction(+:sum)
for (int i = 0; i < n; i++)
    sum+= i/(n/10);

```

1.4 ¿Cómo se compila un código fuente en C para utilizar OpenMP?

Para compilar el código en C con el compilador GNU se debe agregar -fopenmp y exportar

OMP_NUM_THREADS:

```

$ gcc -o omp_hello -fopenmp omp_hello.c
$ export OMP_NUM_THREADS=2
$ ./omp_helloc

```

Y se debe incluir el encabezado omp.h en el código:

```

#include <omp.h>

```

1.5 ¿Cuál función me permite conocer el número de procesadores disponibles para el programa?

Dentro de un programa en C se puede llamar a una función:

```

omp_get_num_procs();

```

1.6 ¿Cómo se define una variable privada en openMP?

En la región paralela de OpenMP una variable puede ser compartida o privada. Si es privada cada hilo en el equipo tiene su propia copia local de la variable.

- Las variables de iteración del loop son privadas por defecto.
- Las variables declaradas dentro de la región paralela son privadas

- Las variables privadas se comportan como variables locales a la región paralela

Se recomienda inicializar las variables privadas dentro de la región paralela pero también se pueden declarar explícitamente dentro de la definición del pragma utilizado para iniciar la región paralela:

```
#pragma omp parallel for private(b)
```

Estas variables son importantes porque son específicas al grupo de hilos por lo que no generan condiciones de carrera. [<http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html>]

1.7 ¿Cómo se definen las variables compartidas en OpenMP?

En la región paralela de OpenMP una variable puede ser compartida o privada. Si es compartida esta variable puede ser accedida por todos los hilos.

- Las variables declaradas fuera de la región paralela usualmente son compartidas
- OpenMP no prevé la condición de carrera para variables compartidas

Las variables se pueden declarar explícitamente dentro de la definición del pragma utilizado para iniciar la región paralela:

```
#pragma omp parallel for shared(n, a)
```

Se debe tener cuidado al actualizar una variable compartida, la condición de carrera debe ser manejada por el programador, si no se tiene este cuidado se pueden tener resultados inesperados.

[<http://jakascorner.com/blog/2016/06/omp-data-sharing-attributes.html>]

Definir ambos tipos de variables lleva a cabo de la siguiente manera:

```
#pragma omp parallel for shared(n, a) private(b)
```

1.8 ¿Para qué sirve *flush* en OpenMP

Al usar hilos, openMP permite que el hilo se vuelve inconsistente con la memoria. La operación *flush* hace que la vista temporal de memoria del hilo sea consistente con memoria y fuerza una orden en las operaciones de memoria de variables explícitas e implícitas. Se puede dar de tres formas:

1.8.1 Strong Flush

Esta operación garantiza la consistencia entre la memoria temporal del hilo y la real. Puede utilizar para garantizar que un segundo hilo lea el valor correcto que escribió el primero, pero el programador es el que se debe encargar que este segundo hilo no haya escrito a la variable desde su último flush:

1. El primer hilo modifica la variable
2. El primer hilo hace flush y actualiza el valor real
3. El segundo hilo hace flush sobre la variable y actualiza su valor temporal
4. El segundo hilo lee el valor de la variable

1.8.2 Release Flush

Esta operación garantiza que lecturas y escrituras a variables compartidas se completen antes de que otro hilo pueda accederla con su propio flush.

1.8.3 Acquire Flush

Esta operación devuelve el valor temporal de una variable compartida al valor que tuvo en su último flush, para que pueda ser sincronizada por un Release Flush. Por lo que ambos permiten que un hilo pueda leer lo

que otro escribió a una variable compartida sin errores de sincronización. Para esto debe ocurrir en orden específico que:

1. El primer hilo escribe a la variable
2. El primer hilo lleva a cabo un release flush
3. El segundo hilo realiza un acquire flush y borra sus cambios temporales
4. El segundo hilo lee el valor de la variable

Las propiedades que definen que tipo de flush se ejecuta no son necesariamente disjuntas. Y un flush puede ser de dos o tres tipos diferentes. [<https://www.openmp.org/spec-html/5.0/openmpsu12.html>]

1.9 ¿Cuál es el propósito de *pragma omp single*?

omp imple especifica que una estructura solo va a ser ejecutada una vez por un thread del equipo. El resto del equipo va a esperar a que está termine a menos de que se especifique **nowait**.

Se puede usar, por ejemplo para que solo un hilo imprima un valor, en vez de que todos lo impriman:

```
#pragma omp single[clause, clause, ... ] new-line
    structured-block
```

Donde "clause" puede ser 1 de las siguientes opciones:

```
copyprivate(list)
firstprivate(list)
nowait(int)
private (int)
```

No se puede utilizar **copyprivate** y **nowait** en el mismo *omp single*.

[https://scc.ustc.edu.cn/zlsc/tc4600/intel/2016.0.109/compiler_c/common/core/GUID-D74F778A-B303-4574-ACE3-0F6A6DC6EFC9.htm]

1.10 ¿Cuáles son tres instrucciones para la sincronización?

Comparación y casos de uso

1.10.1 Regiones críticas

La región crítica restringe la ejecución del bloque asociado a un hilo a la vez. A esta región crítica se le puede asignar un nombre y se considera que todas las regiones sin nombre tienen el mismo nombre no especificado. La sintaxis de una región crítica es:

```
#pragma omp critical [(name) [,] hint(hint-expression)] ] new-line
    structured-block
```

1.10.2 Barreras

La barrera especifica un punto en el que todos los hilos del grupo deben completar las tareas previas a la barrera en la región paralela antes de poder continuar:

[<https://www.openmp.org/spec-html/5.0/openmpsu90.html#x121-4550002.17.2>]

```
#pragma omp barrier new-line
```

Algunas de las restricciones de la barrera son:

- La barrera debe ser encontrada por todos los hilos o ninguno de los hilos
- La secuencia de regiones paralelas y barreras debe ser la misma para todos los hilos

1.10.3 Regiones Ordenadas

Una región ordenada se especifica en un loop compartido, simd o ambos que se ejecuta en el orden natural de iteración del loop. Secuencializa y ordena la ejecución de la región mientras permite que el código fuera de la región se ejecute en paralelo. Tiene dos formas de sintaxis:

```
#pragma omp ordered [clause[ [,] clause] ] new-line
    structured-block
```

Con los posible valores de clause:

```
threads
simd
```

O se puede definir como:

```
#pragma omp ordered clause [[[,] clause] ... ] new-line
```

Con clause:

```
depend(source)
depend(sink : vec)
```

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

Una reducción simplifica algo complejo en algo que es simple. OpenMP se especializa en paralelizar for loops per si hay una variable que se modifica en cada ciclo simplemente paralelizar con:

```
#pragma omp parallel for
```

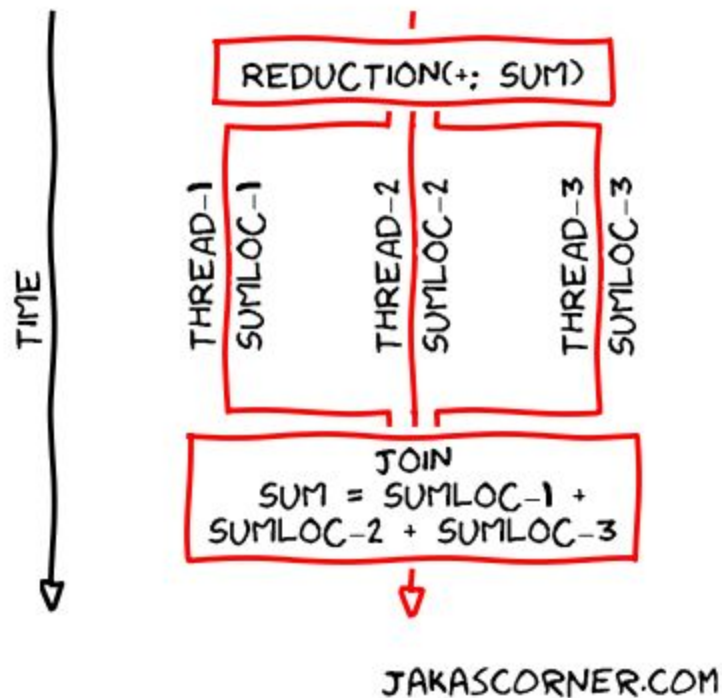
generaría condiciones de carrera. Como este tipo de loops son tan comunes OpenMP provee la función de *reduction* de forma que podemos convertir:

<pre>sum = 0; for (auto i = 0; i < 10; i++) { sum += a[i] }</pre>	<pre>sum = 0; #pragma omp parallel for shared(sum, a) reduction(+: sum) for (auto i = 0; i < 10; i++) { sum += a[i] }</pre>
---	--

Los operadores disponibles para reducción son :

```
+, -, *, &, |, ^, &&, ||
```

Básicamente, openMP crea un equipo de threads que modifica su propia variable y luego une los resultados locales de todas los hilos para obtener el resultado final



[<http://jakascorner.com/blog/2016/06/omp-for-reduction.html>]

2. Análisis

2.1a ¿Qué secciones se pueden paralelizar y cuáles variables pueden ser privadas y cuáles compartidas en pi.c?

La parte paralelizable del programa es el *for loop* que se ejecuta **num_step** veces. Debido al comportamiento de la operación, se requiere utilizar una reducción, ya que **x** depende del número de iteración, esta debe ser privada y similarmente **sum** e **i** serían privadas a cada hilo por la reducción. Las variables **i** se divide para la reducción. **Step**, solo es leída por lo que puede ser compartida, Las variables **pi**, **run_time** y **start_time** no están en la región paralela.

2.2a ¿Qué función realiza `omp_get_wtime()`?

Esta rutina devuelve el valor real del tiempo de ejecución, donde otras formas de llevar el tiempo pueden no contabilizar barreras o hilos detenidos. OpenMP no asegura que el tiempo sea consistente en un set de hilos. La rutina está planeada para utilizarse de la siguiente manera:

```
double start;  
double end;  
start = omp_get_wtime();  
... work to be timed ...
```

```
end = omp_get_wtime();  
printf(~Work took %f seconds\n~, end-start);
```

[<https://www.openmp.org/spec-html/5.0/openmps160.html>]

2.3a Compile y modifique el número de steps

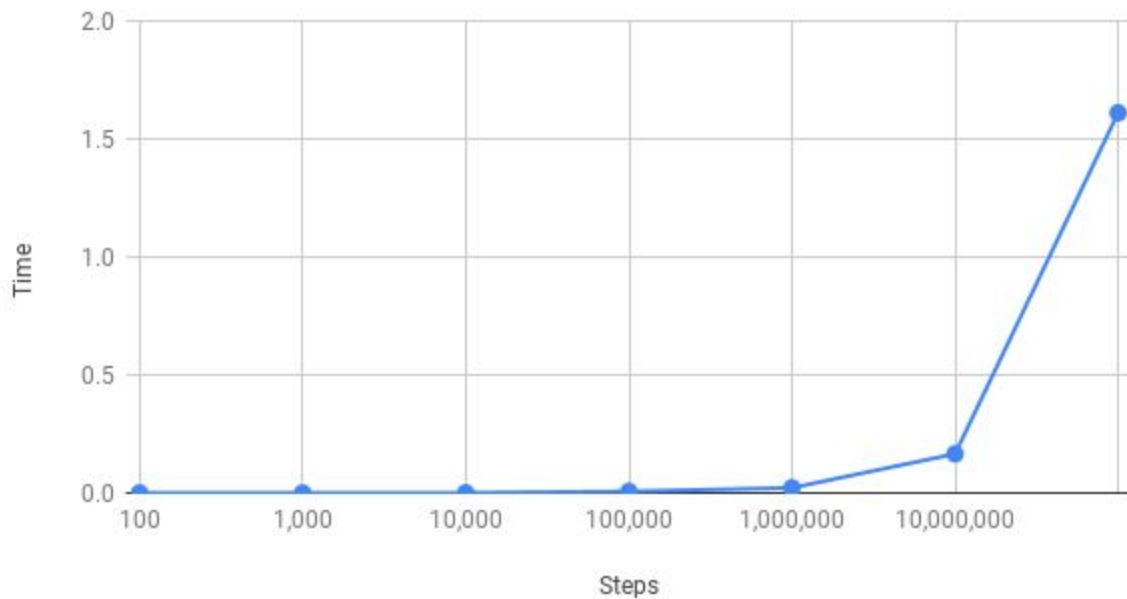
El código se ejecuta con:

```
gcc -o pi -fopenmp pi.c
```

2.4a Grafique el número de pasos contra el tiempo

Se reduce la cantidad de iteraciones hasta que se reduce la precisión del valor obtenido de la función.

Time vs Steps



2.1b Explique los pragmas en pi_loop.c

pragma omp parallel: Define el inicio de la región paralela

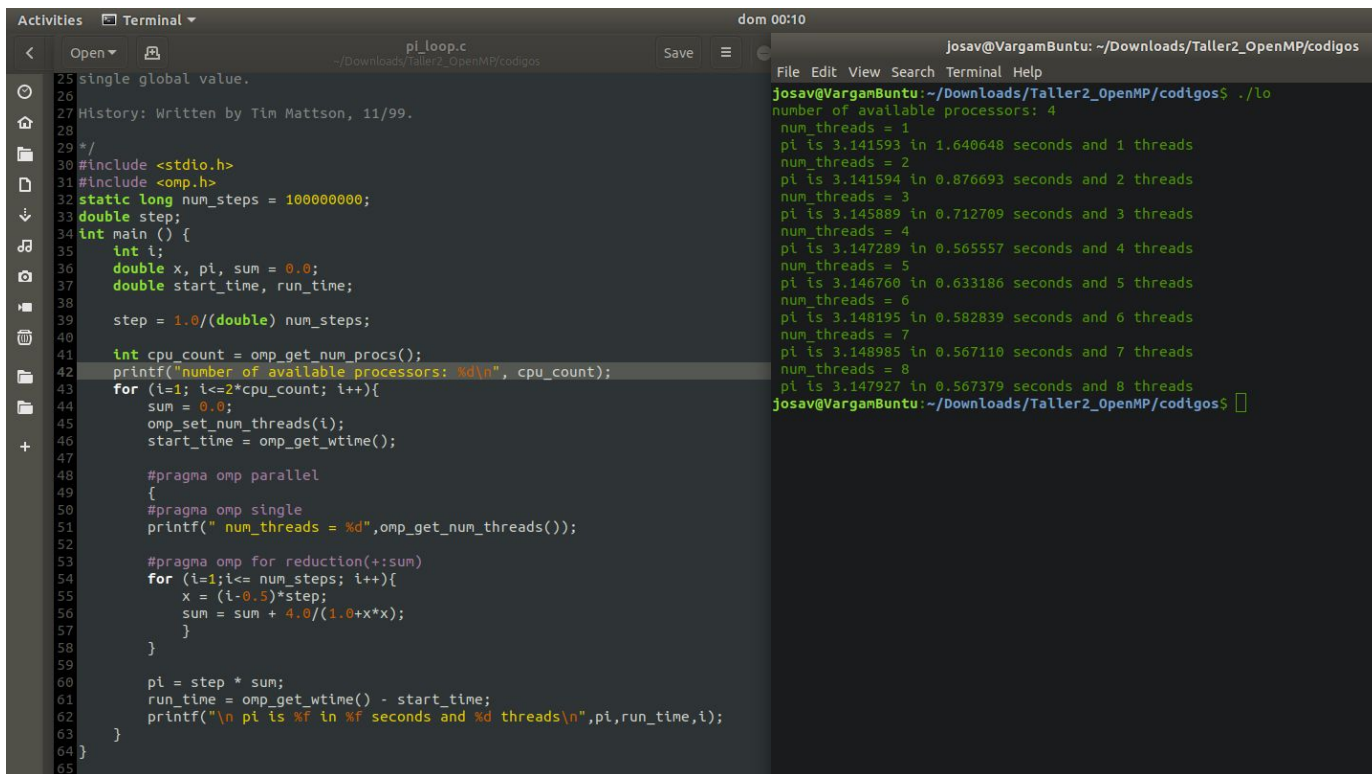
pragma omp single: Hace que la cantidad de hilos solo se imprima una vez

pragma omp for reduction(+:sum): Divide el *for loop* para paralelizar el cálculo de **sum**

2.2b ¿Qué realiza la función `omp_get_num_threads()`?

Para cada iteración, define la cantidad de hilos igual a la iteración. Efectivamente, el *for loop* ejecuta el código paralelizado 4 veces con diferente cantidad de hilos.

2.3b Modifique el programa para que se ejecute con el doble de procesadores disponibles.

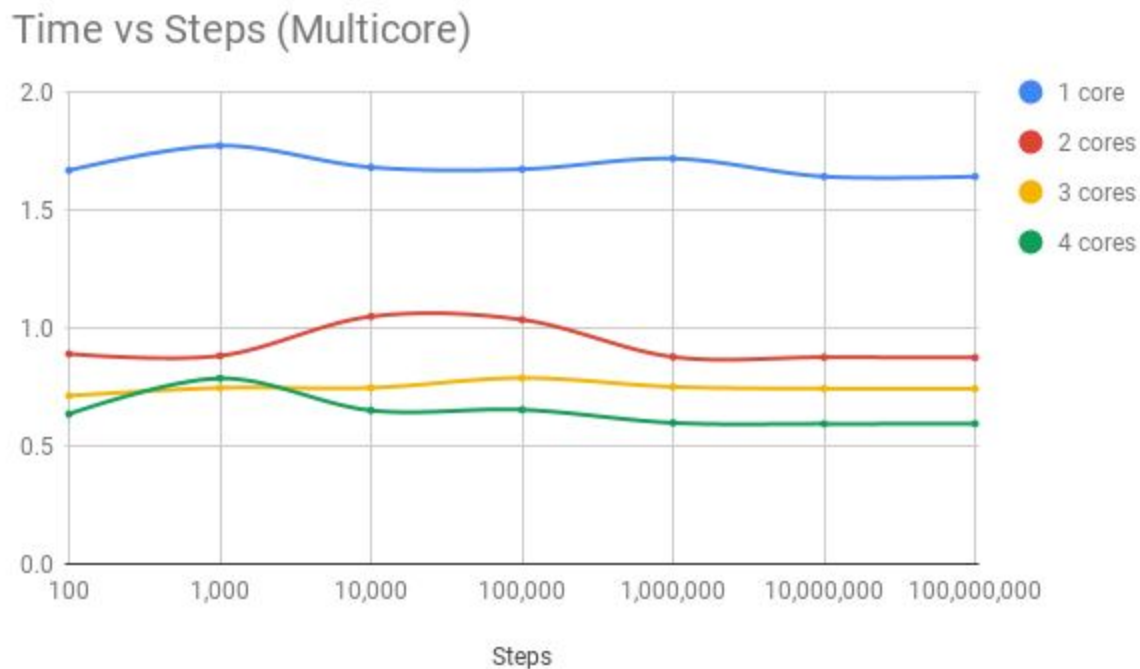


```
25 single global value.
26
27 History: Written by Tim Mattson, 11/99.
28
29 */
30 #include <stdio.h>
31 #include <omp.h>
32 static long num_steps = 100000000;
33 double step;
34 int main () {
35     int i;
36     double x, pi, sum = 0.0;
37     double start_time, run_time;
38
39     step = 1.0/(double) num_steps;
40
41     int cpu_count = omp_get_num_procs();
42     printf("number of available processors: %d\n", cpu_count);
43     for (i=1; i<=2*cpu_count; i++){
44         sum = 0.0;
45         omp_set_num_threads(i);
46         start_time = omp_get_wtime();
47
48         #pragma omp parallel
49         {
50             #pragma omp single
51             printf(" num_threads = %d",omp_get_num_threads());
52
53             #pragma omp for reduction(+:sum)
54             for (i=1;i<= num_steps; i++){
55                 x = (i-0.5)*step;
56                 sum = sum + 4.0/(1.0+x*x);
57             }
58         }
59
60         pi = step * sum;
61         run_time = omp_get_wtime() - start_time;
62         printf("\n pi is %f in %f seconds and %d threads\n",pi,run_time,i);
63     }
64 }
65
```

```
Josav@VargamBuntu: ~/Downloads/Taller2_OpenMP/codigos
number of available processors: 4
num_threads = 1
pi is 3.141593 in 1.640648 seconds and 1 threads
num_threads = 2
pi is 3.141594 in 0.876093 seconds and 2 threads
num_threads = 3
pi is 3.145889 in 0.712709 seconds and 3 threads
num_threads = 4
pi is 3.147289 in 0.565557 seconds and 4 threads
num_threads = 5
pi is 3.146760 in 0.633186 seconds and 5 threads
num_threads = 6
pi is 3.148195 in 0.582839 seconds and 6 threads
num_threads = 7
pi is 3.148985 in 0.567110 seconds and 7 threads
num_threads = 8
pi is 3.147927 in 0.567379 seconds and 8 threads
Josav@VargamBuntu: ~/Downloads/Taller2_OpenMP/codigos
```

2.4b Ejecute el código modificando el parámetro de pasos

2.5b Grafique el número de pasos contra el tiempo. Explique el comportamiento ocurrido.



El tiempo permanece muy constante sin importar las iteraciones. Algunas de las variaciones en los tiempos parecen ser producto de condiciones externas al programa. Las variables se miden en diferentes ejecuciones pero los valores se mantienen siempre en el rango que muestra la figura. De acuerdo a la Ley de Amdahl vemos que la mejora al agregar más núcleos se empieza a reducir y el tiempo de ejecución se acerca más al tiempo de ejecución de la región ni paralelizable.

2.6b Compare los resultados con el ejercicio anterior

Se esperaba que el comportamiento de cada línea fuera más similar al observado en 2.4a, manteniendo la diferencia entre núcleos, pero ya sea por optimizaciones del compilador o diferencias en la programación al agregar OpenMP, no se ve esto ni al forzar la cantidad de hilos con `omp_set_dynamic(0)`

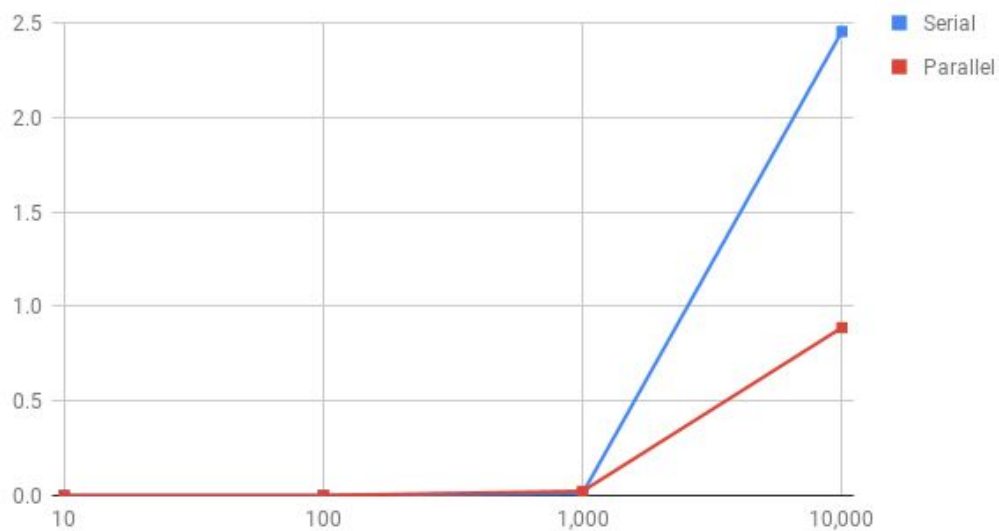
3. Ejercicios prácticos

3.1 SAXPY

SAXPY significa “A por X más Y en precisión simple”, por sus siglas en inglés (Single-Precision A·X Plus Y). Es la función característica de una línea y conlleva multiplicación un escalar con un vector y la suma de dos vectores, ya que X y Y son vectores y A es un escalar. El resultado es un vector. Es una función que se utiliza para mostrar diferencias de sintáxis entre programas.

[<https://devblogs.nvidia.com/six-ways-saxpy/>]

Time vs Steps for saxpy.c



De los resultados se puede ver, (aunque no muy bien en la gráfica) que para valores pequeños (<100) que la versión secuencial tiene mejor rendimiento que la versión paralela, lo que muestra el costo de la paralelización del código.

3.2 Aproximación del número de Euler

```
Activities Terminal ▾
josav@VargamBuntu: ~/Documents/Git/TareasTEC/Arqui 2/Taller 2
File Edit View Search Terminal Help
josav@VargamBuntu:~/Documents/Git/TareasTEC/Arqui 2/Taller 2$ make euler
gcc -o bin/euler -fopenmp euler_s.c
./bin/euler
There are 4 processors available
Euler's number with 1 steps in 0.000021 seconds is: 0.0000000000000000

There are 4 processors available
Euler's number with 10 steps in 0.000125 seconds is: 2.718278769841270

There are 4 processors available
Euler's number with 100 steps in 0.000166 seconds is: 2.718281828459046

There are 4 processors available
Euler's number with 1000 steps in 0.001713 seconds is: 2.718281828459046

There are 4 processors available
Euler's number with 10000 steps in 0.072941 seconds is: 2.718281828459046

There are 4 processors available
Euler's number with 100000 steps in 5.390772 seconds is: 2.718281828459046
```

El valor de la constante de Euler converge muy rápidamente, incluso para 15 cifras significativas por lo que después de 100 iteraciones no se ve un cambio en el valor obtenido.