



Área Académica de Ingeniería en Computadores

Arquitectura de Computadores II

Taller 2

Profesor:

Luis Alonso Barboza Artavia

Estudiante:

Jose Ignacio Granados Marín

Grupo 1

Semestre II 2022

Investigación

1. ¿En qué consiste OpenMP?

OpenMP es un conjunto de directivas de compilador, así como una API para programas escritos en C, C++ o FORTRAN que proporciona soporte para la programación paralela en entornos de memoria compartida. Dicha herramienta identifica las regiones que pueden ejecutarse en paralelo. Los desarrolladores insertan directivas de compilador en su código en regiones paralelas y estas directivas indican a la biblioteca OpenMP que ejecute la región en paralelo [1].

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

Se puede establecer el número de hilos mediante la variable de entorno, con el uso de los siguientes comandos [2]:

- Para el shell bash:

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

- Para el shell csh o tcsh:

```
setenv OMP_NUM_THREADS < number of threads to use >
```

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

Para crear una región paralela basta con codificar la siguiente estructura [3]:

```
#pragma omp parallel  
{  
  
//Parallel region code  
  
}
```

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

El encabezado a incluir es [3]:

```
//OpenMP header  
#include < omp.h >
```

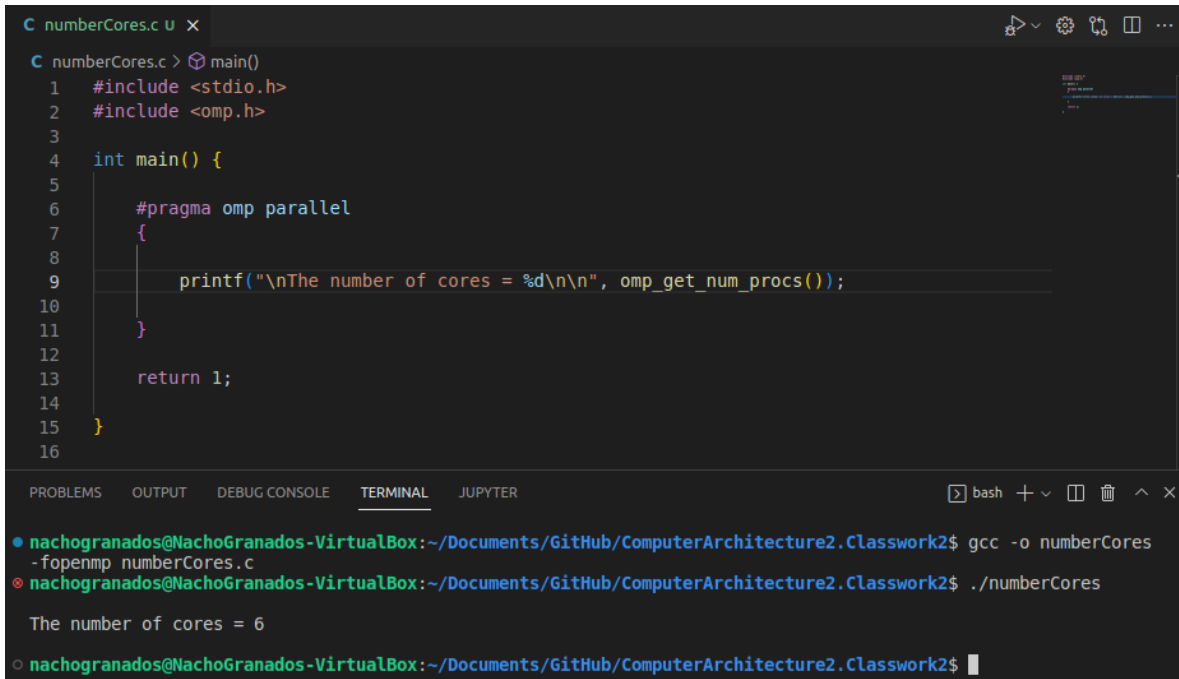
Para compilar un determinado archivo se debe ejecutar el siguiente comando [3]:

```
gcc -o < executable name > -fopenmp < C file >
```

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.

La función que permite conocer el número de procesadores es [4]:

omp_get_num_procs()



```
C numberCores.c U x
C numberCores.c > main()
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main() {
5
6      #pragma omp parallel
7      {
8
9          printf("\nThe number of cores = %d\n", omp_get_num_procs());
10
11      }
12
13      return 1;
14
15  }
16

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$ gcc -o numberCores -fopenmp numberCores.c
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$ ./numberCores

The number of cores = 6
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$
```

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

Las variables privadas son aquellas que se definen de la siguiente forma [5]:

```
#pragma omp parallel private(< variables list >)
{
    //Parallel region code
}
```

Dichas variables son importantes ya que cada hilo tiene su propia copia privada de la variable, y las modificaciones realizadas por un hilo sobre su respectiva copia, no serán visibles para los otros hilos [5].

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

Las variables compartidas son aquellas que se definen de la siguiente forma [5]:

```
#pragma omp parallel shared(< variables list >)
{
    //Parallel region code
}
```

Dichas variables se actualizan por medio del uso de candados o locks ya que, al ser visibles para todos los hilos, se debe garantizar una consistencia en el valor almacenado [5].

8. ¿Para qué sirve flush en OpenMP?

La memoria dispone de una consistencia relajada porque la vista que tenga un hilo, no necesariamente tiene que ser consistente con la memoria en todo momento. Una escritura puede permanecer en la vista temporal del hilo hasta que se vea forzado a escribir en memoria. Del mismo modo, una lectura puede permanecer en la vista temporal del hilo, a menos que se vea obligado a leer desde la memoria. Dado lo anterior, las operaciones de vaciado o flush de OpenMP se utilizan para reforzar la coherencia entre la vista temporal de memoria y memoria de un hilo, o entre la vista de memoria de varios hilos [6].

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

El propósito de la directiva single consiste en especificar que solo un hilo sea el encargado de ejecutar una sección de código. No es necesario que dicho hilo sea el principal [4]. Asimismo, debe utilizar en operaciones que no pueden ser ejecutadas en forma paralela o simplemente porque cierta sección del código no es paralilizable.

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

Algunas directivas de sincronización son las siguientes [7]:

- Crítico (Critical) es una directiva de exclusión mutua que garantiza que solo un hilo pueda ingresar a una región crítica a la vez. Dicha instrucción se puede utilizar cuando no se requiera que múltiples accedan o hagan uso de un recurso compartido simultáneamente.

- Atómica (Atomic) es una directiva de exclusión mutua, similar a critical, pero solo se aplica a la actualización de una ubicación de memoria. Por su parte, la sección crítica es muy general y siempre hay una sobrecarga cada vez que un hilo entra y sale de la sección crítica. Por el contrario, una operación atómica tiene una sobrecarga mucho menor y se basa en el hardware, para realizar la operación atómica. Dicha instrucción se puede utilizar cuando solo es necesario realizar lecturas o escrituras a una determinada sección de la memoria.
- Barrera (Barrier) es una directiva que permite que los hilos de una región paralela esperen hasta que todos los demás hilos de esa sección alcancen el mismo punto. De esta manera, la ejecución, luego de la barrera, continúa en paralelo. Dicha instrucción se puede utilizar cuando una determinada sección de código requiere ciertos cálculos anteriores realizados por cada uno de los hilos y no puede ser ejecutado sin ellos.

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

Las cláusulas de reducción son directivas de atributos de uso compartido de datos, que se pueden utilizar para realizar cálculos recurrentes en paralelo. Las cláusulas de reducción incluyen cláusulas de alcance de reducción y cláusulas de participación de reducción. Las primeras definen la región en la que se calcula una reducción. Las segundas definen a los participantes en la reducción. Dichos identificadores se definen de la siguiente manera [8]:

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

Análisis

Primer código pi.c:

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

Posibles secciones a paralelizar:

- Cálculo de la variable start_time.
- Ciclo for.
- Cálculo de la variable run_time.

En este caso en particular, las secciones anteriores pueden ser ejecutadas, en forma paralela, por cada uno de los hilos disponibles para obtener el resultado más rápido, dado que el ciclo debe realizar una gran cantidad de iteraciones.

Posibles variables compartidas:

- num_steps.
- i.
- pi.
- sum.
- step.

Las variables anteriores pueden ser visibles y modificadas para cada uno de los hilos ya que son valores que cada hilo por separado necesita conocer para ejecutar los cálculos correctos.

Posibles variables privadas:

- x.
- start_time.
- run_time.

A diferencia de las variables compartidas, las variables anteriores deben ser de uso exclusivo para cada hilo, ya que cada uno de ellos realizará sus respectivos cálculos y aportará al resultado final sin que otro hilo afecte su ejecución.

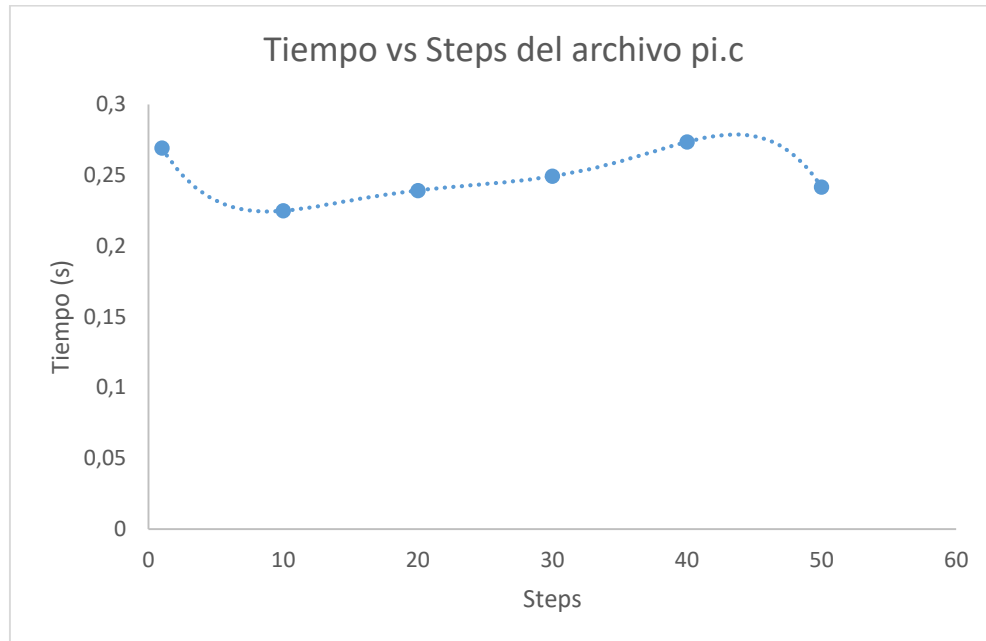
2. ¿Qué realiza la función `omp_get_wtime()`?

La función `omp_get_wtime()` devuelve un valor de precisión doble igual al número de segundos desde el valor inicial del reloj en tiempo real del sistema operativo. Se garantiza que el valor inicial no cambiará durante la ejecución del programa [9].

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

```
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$ gcc -o pi -fopenmp pi.c
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$ ./pi
pi with 100000000 steps is 3.141593 in 0.269233 seconds
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$
```

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



El comportamiento obtenido en el gráfico anterior demuestra que incrementos constantes mejoran el tiempo de ejecución, pero no en gran medida. Si se compara el tiempo obtenido de 1 step con 10, 20, 30 o 50 steps, se puede observar una reducción del mismo, lo cual demuestra que dichos pasos sí reducen el tiempo de ejecución. Sin embargo, si se compara el tiempo obtenido de 1 step con 40 steps, la mejora es mínima, indicando de esta manera que dicho número de pasos no beneficia el cálculo aproximado de la constante π .

Segundo código pi_loop.c:

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

- `#pragma omp parallel`

Esta definición pretende que cada hilo ejecute el pragam `omp single` y el `pragma omp for`, de tal manera que, cada uno de estos procesos aporte al cálculo final de la aproximación del valor π .

- `#pragma omp single`

Esta definición pretende que un único hilo sea el encargado de obtener e imprimir el número de hilos del programa.

- `#pragma omp for`

Esta definición pretende que el trabajo del ciclo iterativo sea dividido entre cada uno de los hilos disponibles del programa.

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

La función `omp_get_num_threads()` devuelve el número de subprocesos del equipo que ejecuta actualmente la región paralela desde la que se llama. La función se enlaza a la directiva `PARALLEL` que la encierra más cercana [10].

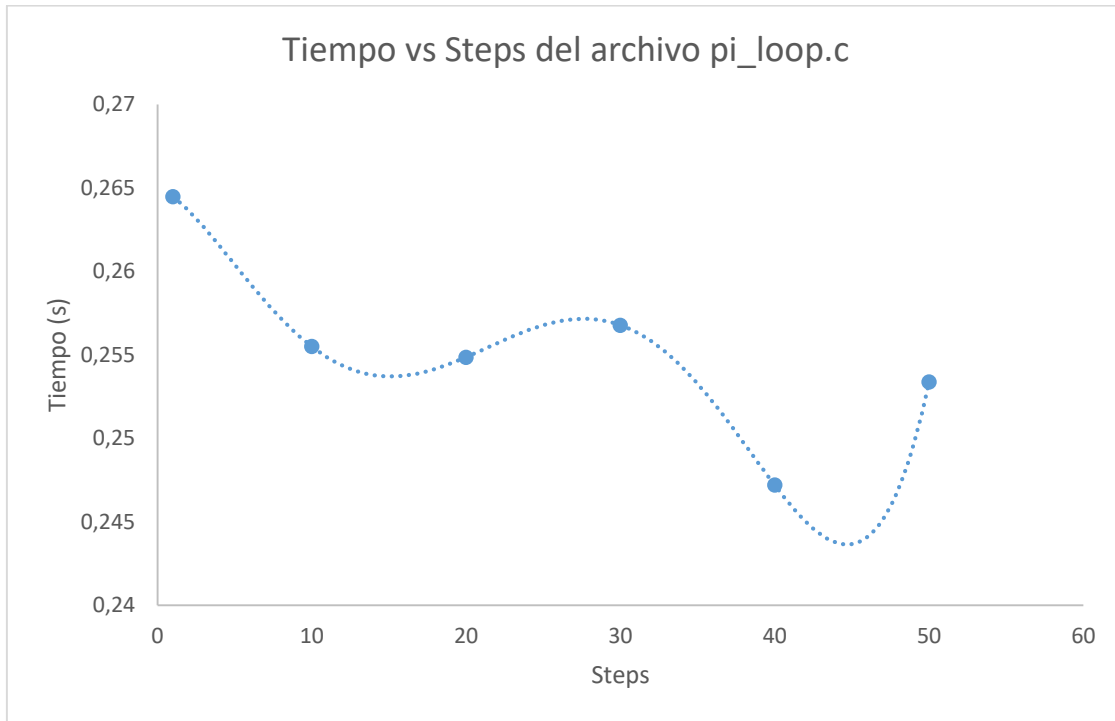
3. En la línea 41, el ciclo se realiza 4 veces. Realice un cambio en el código fuente para que el ciclo se repita el doble de la cantidad de procesadores disponibles para el programa. Incluya un screenshot con el cambio.

```
for(i = 1; i <= 2 * omp_get_num_procs(); i++) {  
  
    sum = 0.0;  
  
    omp_set_num_threads(i);
```

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

```
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$ gcc -o pi_loop -fopenmp pi_loop.c  
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$ ./pi_loop  
num_threads = 1  
pi is 3.141593 in 0.264478 seconds and 1 threads  
num_threads = 2  
pi is 3.141593 in 0.145351 seconds and 2 threads  
num_threads = 3  
pi is 3.141593 in 0.118653 seconds and 3 threads  
num_threads = 4  
pi is 3.141593 in 0.093803 seconds and 4 threads  
num_threads = 5  
pi is 3.141593 in 0.105455 seconds and 5 threads  
num_threads = 6  
pi is 3.141593 in 0.103408 seconds and 6 threads  
num_threads = 7  
pi is 3.141593 in 0.095723 seconds and 7 threads  
num_threads = 8  
pi is 3.141593 in 0.126656 seconds and 8 threads  
num_threads = 9  
pi is 3.141593 in 0.087392 seconds and 9 threads  
num_threads = 10  
pi is 3.141593 in 0.124602 seconds and 10 threads  
num_threads = 11  
pi is 3.141593 in 0.103441 seconds and 11 threads  
num_threads = 12  
pi is 3.141593 in 0.081429 seconds and 12 threads  
nachogranados@NachoGranados-VirtualBox:~/Documents/GitHub/ComputerArchitecture2.Classwork2$
```


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



El comportamiento obtenido en el gráfico anterior demuestra que a mayor número de steps, menor será el tiempo de ejecución en comparación con un step de 1. Se puede observar que el step de 30 generó un pequeño incremento en el tiempo de ejecución en comparación con los steps de 10 y 20, lo que indica que el cálculo no es óptimo con dicho valor. Por otra parte, se puede notar, entre los steps 30, 40 y 50, una forma parabólica donde el tiempo de ejecución tiende a disminuir desde el step 30 hasta el 45 aproximadamente, pero luego, cambia dicha conducta por una ascendente, afectando la duración del programa. Por lo que, el step 50, al igual que el 30, no generan un alto rendimiento en el cálculo aproximado de la constante pi.

6. Compare los resultados con el ejercicio anterior.

Al comparar las 2 gráficas anteriores se puede observar que la primera posee un comportamiento más estable, menos fluctuante o relativamente constante en comparación con la segunda y esto se debe principalmente a la complejidad entre un algoritmo y otro. Sin embargo, en ambas representaciones se puede notar existen ciertos valores de steps que afectan o incrementan el tiempo de ejecución, demostrando que los algoritmos no se ejecutarán más rápido a mayor número de steps.

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.

SAXPY serial:

```
1 #include <stdio.h>
2 #include <time.h>
3
4 int main() {
5
6     int i;
7
8     //int n = 10;
9     //int n = 1000;
10    int n = 100000;
11
12    float a = 2.0;
13
14    float x[n];
15    float y[n];
16
17    clock_t start_time, end_time;
18
19    for(i = 0; i < n; i++) {
20
21        x[i] = 1.0;
22        y[i] = 2.0;
23    }
24
25    start_time = clock();
26
27    for(i = 0; i < n; i++) {
28
29        y[i] = a*x[i] + y[i];
30    }
31
32    end_time = clock();
33
34    printf ("SAXPY time: %f\n", (double)(end_time - start_time)/ CLOCKS_PER_SEC);
35
36    return 0;
37
38 }
39
40 }
```

SAXPY paralelo:

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <omp.h>
4
5 int main() {
6
7     int i;
8
9     //int n = 10;
10    //int n = 1000;
11    int n = 1000000;
12
13    float a = 2.0;
14
15    float x[n];
16    float y[n];
17
18    double start_time;
19    double end_time;
20
21    for (i = 0; i < n; i++) {
22
23        x[i] = 1.0;
24        y[i] = 2.0;
25
26    }
27
28    start_time = omp_get_wtime();
29
30    #pragma omp parallel for private(i)
31    for (i = 0; i < n; i++) {
32
33        y[i] = a*x[i] + y[i];
34
35    }
36
37    end_time = omp_get_wtime();
38
39    printf ("SAXPY time: %f\n", end_time - start_time);
40
41    return 0;
42
43 }
```

Tabla comparativa:

Tiempo de ejecución			
Tipo Ejecución	Valores de n		
	n = 10	n = 1000	n = 100000
Serial	0,000001	0,000004	0,000219
Paralela	0,000203	0,000215	0,000302

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.

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <omp.h>
4
5 int main() {
6
7     double e;
8
9     int i;
10
11     //int n = 165;
12     //int n = 330;
13     //int n = 495;
14     //int n = 660;
15     //int n = 825;
16     int n = 1000;
17
18     double a[1000];
19
20     double start_time;
21     double end_time;
22
23     a[0] = 1.0;
24
25     start_time = omp_get_wtime();
26
27     #pragma omp parallel for private(i)
28     for (i = 1; i < n; i++) {
29
30         a[i] = a[i-1] / i;
31
32     }
33
34     e = 1.0;
35
36     #pragma omp parallel for private(i)
37     for (i = n - 1; i > 0; i--) {
38
39         e += a[i];
40
41     }
42
43     end_time = omp_get_wtime();
44
45     printf("Euler constant e = %.16lf\n", e);
46
47     printf ("Time: %f\n", end_time - start_time);
48
49     return 0;
50
51 }
```

Tabla comparativa:

Valor de n	Valor aproximado	Tiempo de ejecución
n = 500	2.7182818284590455	0.005958
n = 600	2.7182818284590455	0.004879
n = 700	2.7182818284590455	0.000498
n = 800	2.7182818284590455	0.000513
n = 900	2.7182818284590455	0.000661
n = 1000	2.7182818284590455	0.000532
Valor real	2.7182818284590452	-

Referencias

- [1] Chakraborty, A. (2019, octubre 11). *What is OpenMP?*. Recuperado 27 de septiembre de 2022, de <https://www.tutorialspoint.com/what-is-openmp>
- [2] *Setting the Number of Threads Using an OpenMP* Environment Variable*. (s. f.). Recuperado 27 de septiembre de 2022, de http://portal.nacad.ufrj.br/online/intel/mkl/common/mkl_userguide/GUID-B13E207A-FC51-4973-9676-B40545599272.htm
- [3] GeeksforGeeks. (2021, junio 30). *OpenMP / Hello World program*. Recuperado 27 de septiembre de 2022, de <https://www.geeksforgeeks.org/openmp-hello-world-program/>
- [4] Microsoft. (2022, 26 septiembre). *Funciones de OpenMP*. Microsoft Learn. Recuperado 27 de septiembre de 2022, de <https://learn.microsoft.com/es-cpp/parallel/openmp/reference/openmp-functions?view=msvc-170#omp-get-num-procs>
- [5] Xu, Z. (2016). *Lecture 12: Introduction to OpenMP (Part 1)*. University of Notre Dame. <https://www.ibm.com/docs/en/zos/2.2.0?topic=programs-shared-private-variables-in-parallel-environment>
- [6] *The Flush Operation*. (2018, noviembre). Recuperado 27 de septiembre de 2022, de <https://www.openmp.org/spec-html/5.0/openmps12.html>
- [7] Open MP Beat. (2013, 23 octubre). *OpenMP Synchronization Concepts*. Recuperado 27 de septiembre de 2022, de <https://openmpbeat.wordpress.com/2013/10/23/openmp-synchronization/>
- [8] *Reduction Clauses and Directives*. (2018, noviembre). Recuperado 27 de septiembre de 2022, de <https://www.openmp.org/spec-html/5.0/openmps107.html>
- [9] IBM. (2021, marzo 6). *omp_get_wtime()*. © Copyright IBM Corporation 1990, 2015. Recuperado 28 de septiembre de 2022, de <https://www.scribbr.es/citar/generador/folders/7Ijug2fNdaZS56zInfwbbz/lists/5swWyM3THBc7BKUaSnPDi5/citar/pagina-web/>
- [10] *omp_get_num_threads()*. (2021, marzo 3). © Copyright IBM Corporation 1990, 2015. Recuperado 28 de septiembre de 2022, de <https://www.ibm.com/docs/en/xl-fortran-aix/15.1.3?topic=openmp-omp-get-num-threads>
- [11] Pacheco, A. (2018, 17 julio). *Introduction to OpenMP*. Lehigh University. Recuperado 28 de septiembre de 2022, de <https://www.lehigh.edu/~alp514/hpc2017/openmp.pdf>
- [12] Rosetta Code. (2022, 18 septiembre). *Calculating the value of e*. Recuperado 28 de septiembre de 2022, de https://rosettacode.org/wiki/Calculating_the_value_of_e#C