



**Área Académica de Ingeniería en Computadores**

**CE-4302 Arquitectura de Computadores II**

**Profesor:**

Ronald García Fernández

**Estudiante:**

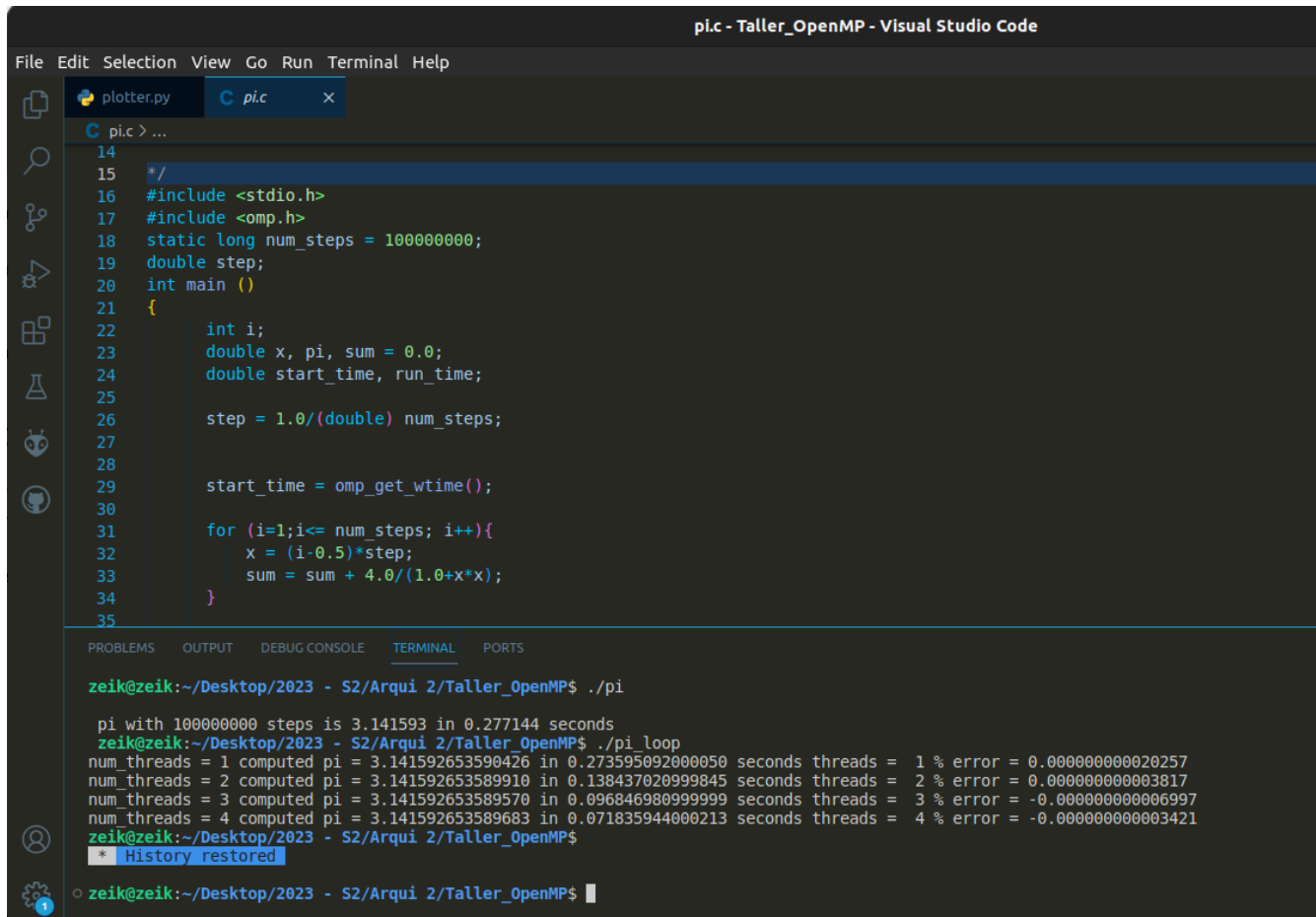
Roy Acevedo Méndez

**Taller OpenMP**

**II Semestre 2023**

# Análisis de resultados

## Apartado 3



```
pi.c - Taller_OpenMP - Visual Studio Code
File Edit Selection View Go Run Terminal Help
plotter.py C pi.c x
C pi.c > ...
14
15 */
16 #include <stdio.h>
17 #include <omp.h>
18 static long num_steps = 100000000;
19 double step;
20 int main ()
21 {
22     int i;
23     double x, pi, sum = 0.0;
24     double start_time, run_time;
25
26     step = 1.0/(double) num_steps;
27
28
29     start_time = omp_get_wtime();
30
31     for (i=1; i<= num_steps; i++){
32         x = (i-0.5)*step;
33         sum = sum + 4.0/(1.0+x*x);
34     }
35
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
zeik@zeik:~/Desktop/2023 - S2/Arqui 2/Taller_OpenMP$ ./pi
pi with 100000000 steps is 3.141593 in 0.277144 seconds
zeik@zeik:~/Desktop/2023 - S2/Arqui 2/Taller_OpenMP$ ./pi_loop
num_threads = 1 computed pi = 3.141592653590426 in 0.273595092000050 seconds threads = 1 % error = 0.000000000020257
num_threads = 2 computed pi = 3.141592653589910 in 0.138437020999845 seconds threads = 2 % error = 0.00000000003817
num_threads = 3 computed pi = 3.141592653589570 in 0.096846980999999 seconds threads = 3 % error = -0.00000000006997
num_threads = 4 computed pi = 3.141592653589683 in 0.071835944000213 seconds threads = 4 % error = -0.00000000003421
zeik@zeik:~/Desktop/2023 - S2/Arqui 2/Taller_OpenMP$
* History restored
zeik@zeik:~/Desktop/2023 - S2/Arqui 2/Taller_OpenMP$
```

Figura 1. Ejecución de ambos programas pi.

En la **Figura 1** se muestra la ejecución de ambos algoritmos de cálculo de pi, tanto en single threading como multi-threading, con un valor estandar de 4 threads para la versión multi-threading.

```

32 static long num_steps = 100000000;
33 static double reference_pi = 3.14159265358979;
34 double step;
35 int main () {
36     int i;
37     double x, pi, sum = 0.0;
38     double start_time, run_time;
39
40     int max_threads = omp_get_max_threads();
41
42     step = 1.0/(double) num_steps;
43     for (i = 1; i <= max_threads; i++) {
44         sum = 0.0;
45         omp_set_num_threads(i);
46         start_time = omp_get_wtime();
47     #pragma omp parallel
48     {
49         #pragma omp single
50         printf("num_threads = %d ",omp_get_num_threads());
51     }

```

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
● zeik@zeik:~/Desktop/2023 - S2/Arqui 2/Taller_OpenMP$ ./pi_loop
num_threads = 1 computed pi = 3.141592653590426 in 0.278947291999430 seconds threads = 1 % error = 0.000000000020257
num_threads = 2 computed pi = 3.141592653589910 in 0.142235454000911 seconds threads = 2 % error = 0.00000000003817
num_threads = 3 computed pi = 3.141592653589570 in 0.096577811000316 seconds threads = 3 % error = -0.00000000006997
num_threads = 4 computed pi = 3.141592653589683 in 0.074410020999494 seconds threads = 4 % error = -0.00000000003421
num_threads = 5 computed pi = 3.141592653589737 in 0.073637539999254 seconds threads = 5 % error = -0.00000000001696
num_threads = 6 computed pi = 3.141592653589646 in 0.060919248999198 seconds threads = 6 % error = -0.00000000004594
num_threads = 7 computed pi = 3.141592653589740 in 0.052855113000987 seconds threads = 7 % error = -0.00000000001583
num_threads = 8 computed pi = 3.141592653589816 in 0.046265535000202 seconds threads = 8 % error = 0.00000000000820
num_threads = 9 computed pi = 3.141592653589565 in 0.056846780999897 seconds threads = 9 % error = -0.00000000007153
num_threads = 10 computed pi = 3.141592653589624 in 0.046695842998815 seconds threads = 10 % error = -0.000000000005273
num_threads = 11 computed pi = 3.141592653589843 in 0.058980648000215 seconds threads = 11 % error = 0.00000000001682
num_threads = 12 computed pi = 3.141592653589828 in 0.053275473001122 seconds threads = 12 % error = 0.00000000001216
○ zeik@zeik:~/Desktop/2023 - S2/Arqui 2/Taller_OpenMP$

```

Figura 2. Ejecución con threads del sistema.

En la **Figura 2** se aprecian los resultados después de haber realizado las modificaciones al código de la versión multi-threading, donde se usó `omp_get_max_threads()` para obtener el número máximo de threads del sistema, que en mi caso fueron 12 threads.

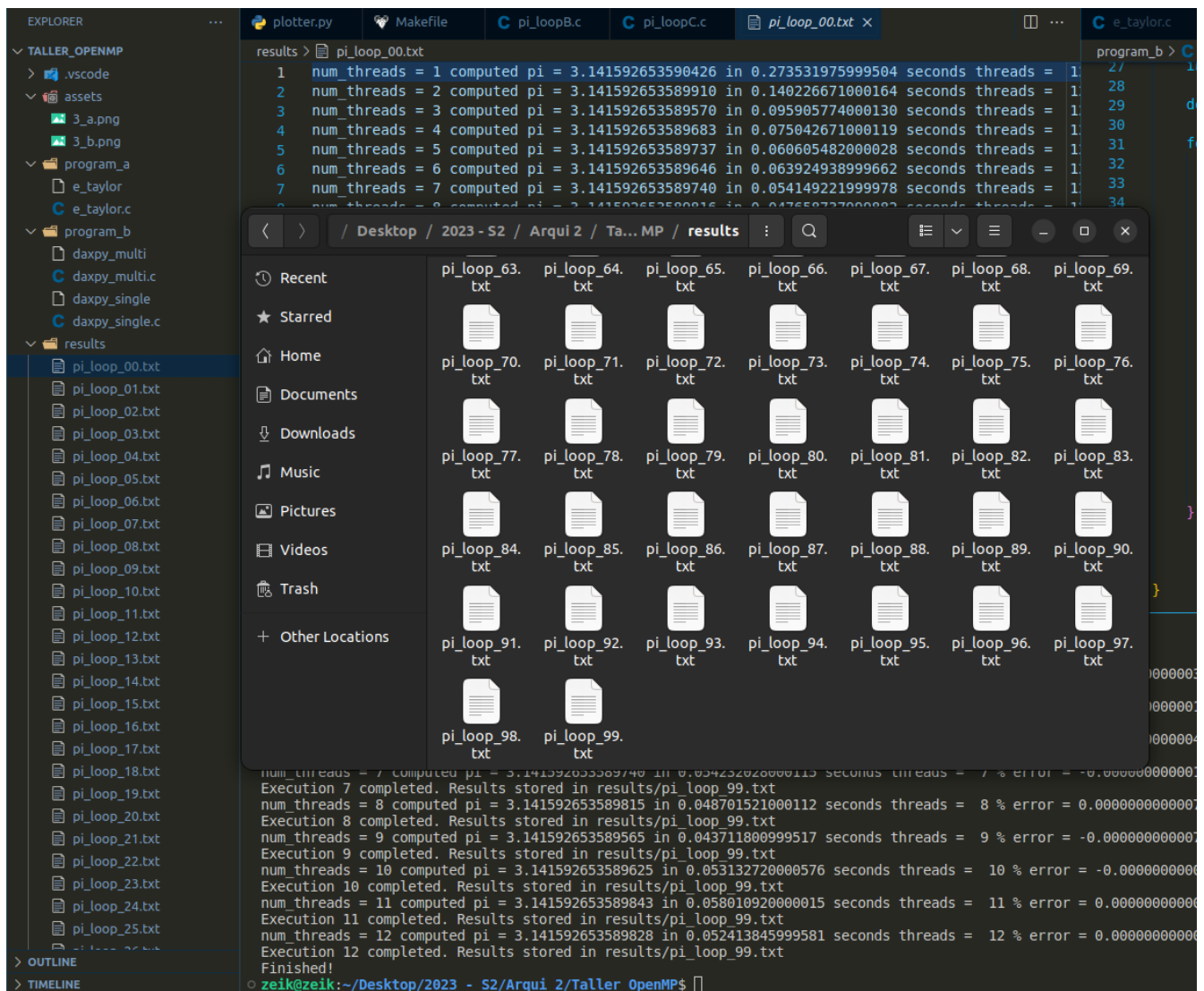
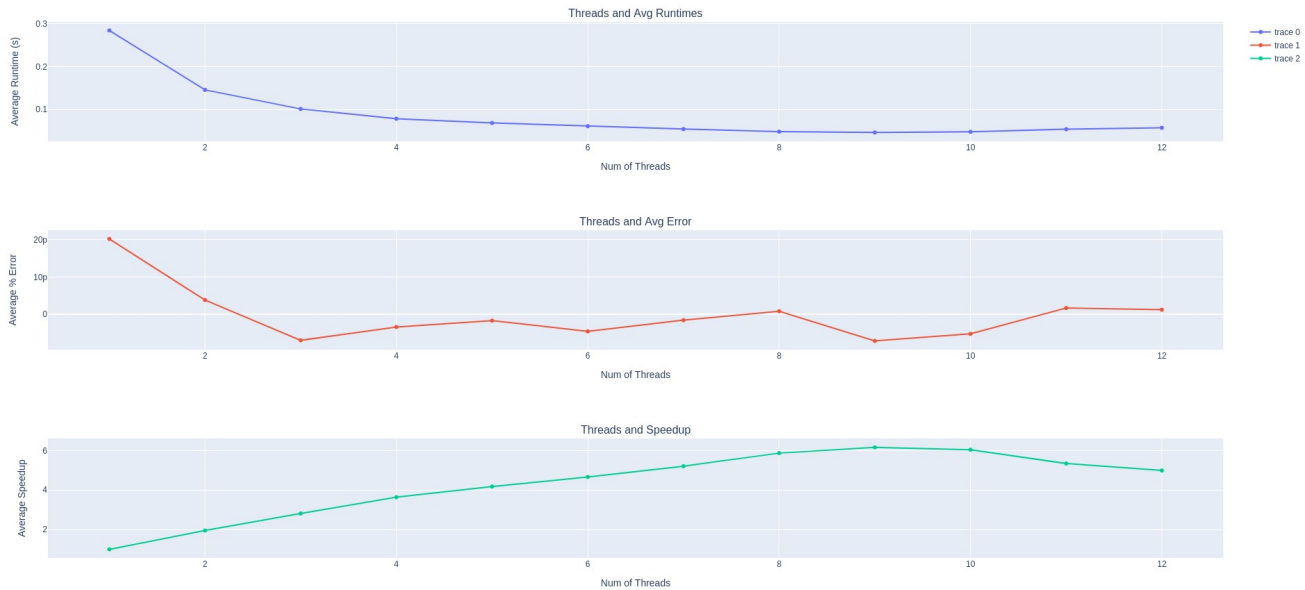


Figura 3. Ejecución múltiple del código pi\_loop.

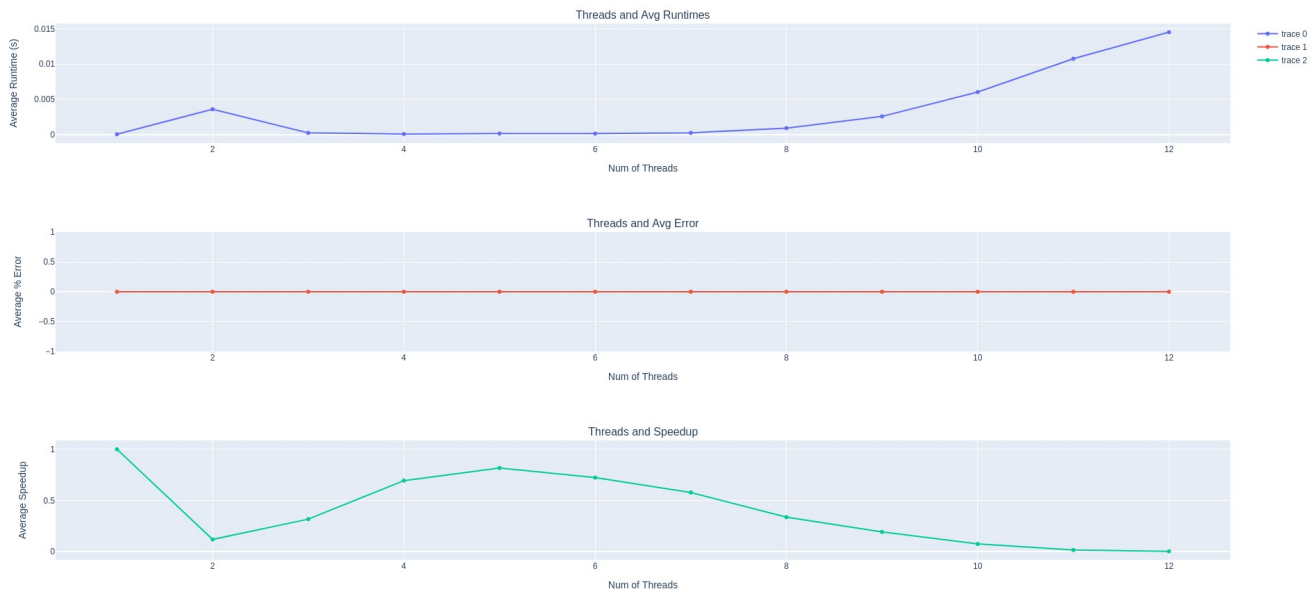
En la **Figura 3** se modificó el código para poder guardar los resultados de las ejecuciones en un archivo de texto con su respectivo identificador, cada archivo de texto contiene los resultados en base al número de threads del sistema, en mi caso, de 1 a 12 como máximo. El programa se ejecutó unas 100 veces y se guardó en la carpeta de results.



**Figura 4. Resultados gráficos del algoritmo de pi multi-threading.**

En la **Figura 4** se aprecian los resultados graficos de la versión multi-threading del programa de pi\_loop después de las modificaciones para adaptarse a los threads del sistema. Se puede ver como el tiempo de ejecución en promedio disminuye conforme se incrementa el número de threads, llegando a punto mínimo en los 9 threads. También se observa el speed up del programa donde si bien aumenta con el número de threads, este llega a un punto donde ya no incrementa más, y por el contrario empieza a disminuir así le pongamos muchos más threads.

## Apartado 4



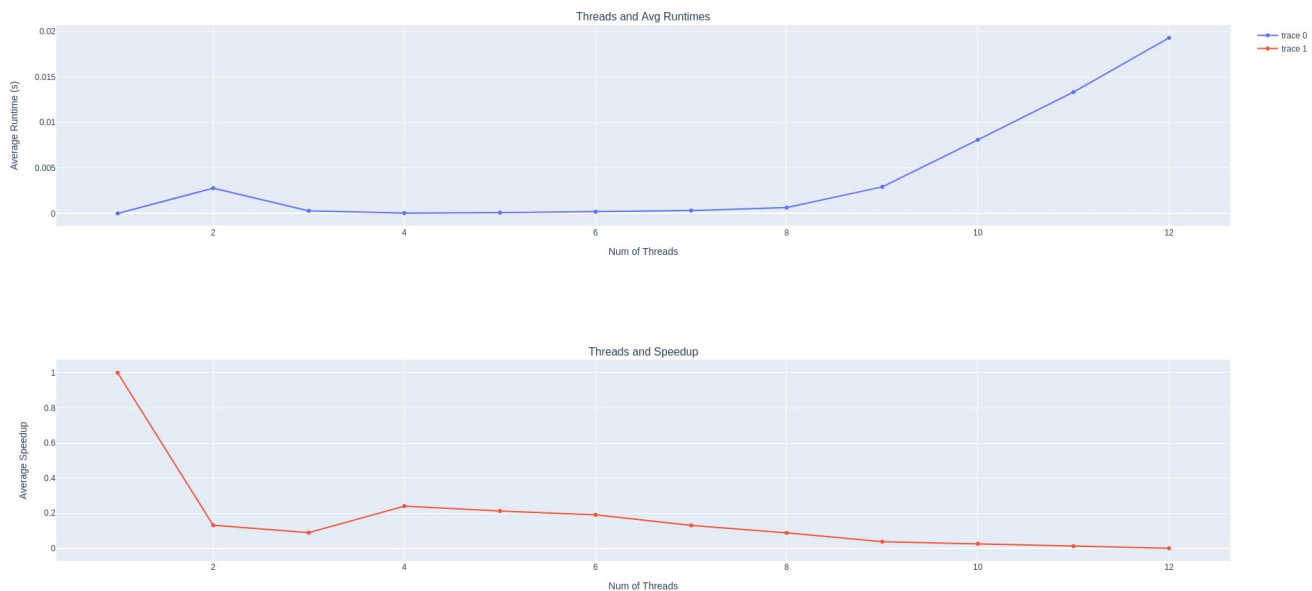
**Figura 5. Resultados gráficos del programa a.**

En la **Figura 5**, observan las gráficas en relación al programa a del cálculo de la constante 'e' mediante series de Taylor para la versión multi-threading. Para este programa, se obtuvo un porcentaje de error muy cercano al 0%, lo cual se debe a la casi nula variación entre los resultados de la constante 'e' obtenida por el programa y el valor tomado como referencia, además, de que para la cantidad de términos 'n' como aproximaciones, se usó un valor relativamente alto, 100, para así poder aproximar más el valor de la constante, que ya en si a partir de 10 aproximaciones ya el valor es muy cercano al de referencia. Lo que también se observó, fue como el speed up fue disminuyendo con forme se incrementaba los threads y a apartir de 12 threads, el promedio de tiempo de ejecución era algo mayor al de sus iteraciones anteriores con menos threads. La cantidad de paralelismo que se puede lograr en un programa está limitada por la naturaleza del algoritmo y la carga de trabajo. Si el cálculo de la serie de Taylor para 'e' no presenta suficiente paralelismo intrínseco o la carga de trabajo por iteración es demasiado pequeña, agregar más hilos puede no proporcionar beneficios significativos y podría introducir sobrecarga debido a la gestión de hilos.

Al ser un algoritmo algo sencillo, si la carga de trabajo por cada hilo es muy pequeña, es posible que no obtengamos una mejora significativa al aumentar el número de hilos. Se tendría que asegurar que la cantidad de trabajo en cada iteración del bucle sea lo suficientemente grande como para que la paralelización sea beneficiosa. También hay que tener en cuenta que para obtener resultados precisos, especialmente con una gran cantidad

de términos en la serie de Taylor, puede ser necesario utilizar tipos de datos de punto flotante de alta precisión, como `long double`.

Para este programa se usó también el `reduction(+:e)`, esto para poder otorgarle un sección a cada thread para que la trabaje y al final realizar la unión de los resultados, también `private(factorial)` para que cada thread tuviera una instancia de esa variable y la trabajara de manera independiente sin afectar las el valor de los otros threads.



**Figura 6. Resultados gráficos del programa b de DAXPY.**

Según los resultados de la **Figura 6**, se ve un aumento en el tiempo de ejecución justo al ejecutar la paralelización con el número máximo de threads(12). Puede que exista un overhead cuando se ejecuta un programa en múltiples hilos, ya que está asociado con la creación, gestión y sincronización de los hilos. Este overhead puede volverse más significativo cuando se utiliza el número máximo de hilos, lo que puede ralentizar el programa en comparación con la ejecución en un número menor de hilos. Aunque la paralelización de un programa puede mejorar el rendimiento en muchos casos, no siempre es beneficioso utilizar el número máximo de hilos, y puede haber situaciones en las que el rendimiento empeore debido a diversos factores.

## Referencias

- Microsoft. (s.f.). (2023). OpenMP Directives. Microsoft Docs.  
<https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170>
- [CoffeeBeforeArch](#) (2023) <https://www.youtube.com/watch?v=gW9EiEQAkDU&t=46s>
- (2023) <https://www.stolaf.edu/people/rab/pdc/text/alg.htm>