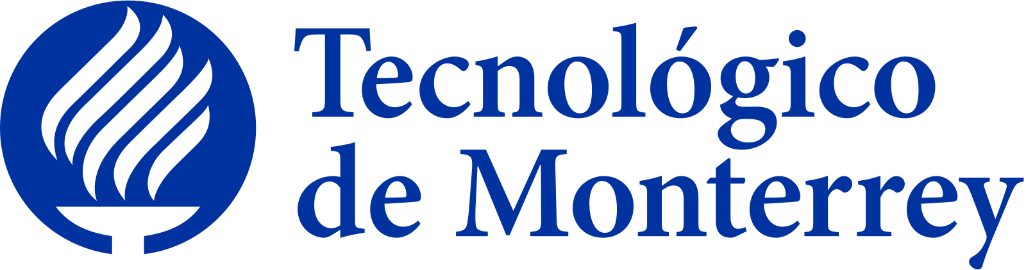
**Maestría en Inteligencia Artificial Aplicada**

**Cómputo en la nube**



**Maestro: Dr. Eduardo Antonio Cendejas Castro**

**Tutor: Julio César Salgado Ramírez**

**Tarea 1. Programación de una solución paralela**

César Julián Camarillo Pérez A01795688

26 de enero 2025

**Introducción**

La computación paralela nos permite ejecutar ciertas operaciones al mismo tiempo en una sola instancia, lo que nos permite poder mejorar el desempeño de la computadora que ejecuta las tareas de esta forma, así como mejorar en los tiempos de procesamiento, la velocidad, los costos, estos relacionados al ahorro en el consumo de energía al encontrar mayor eficiencia en los procesos.

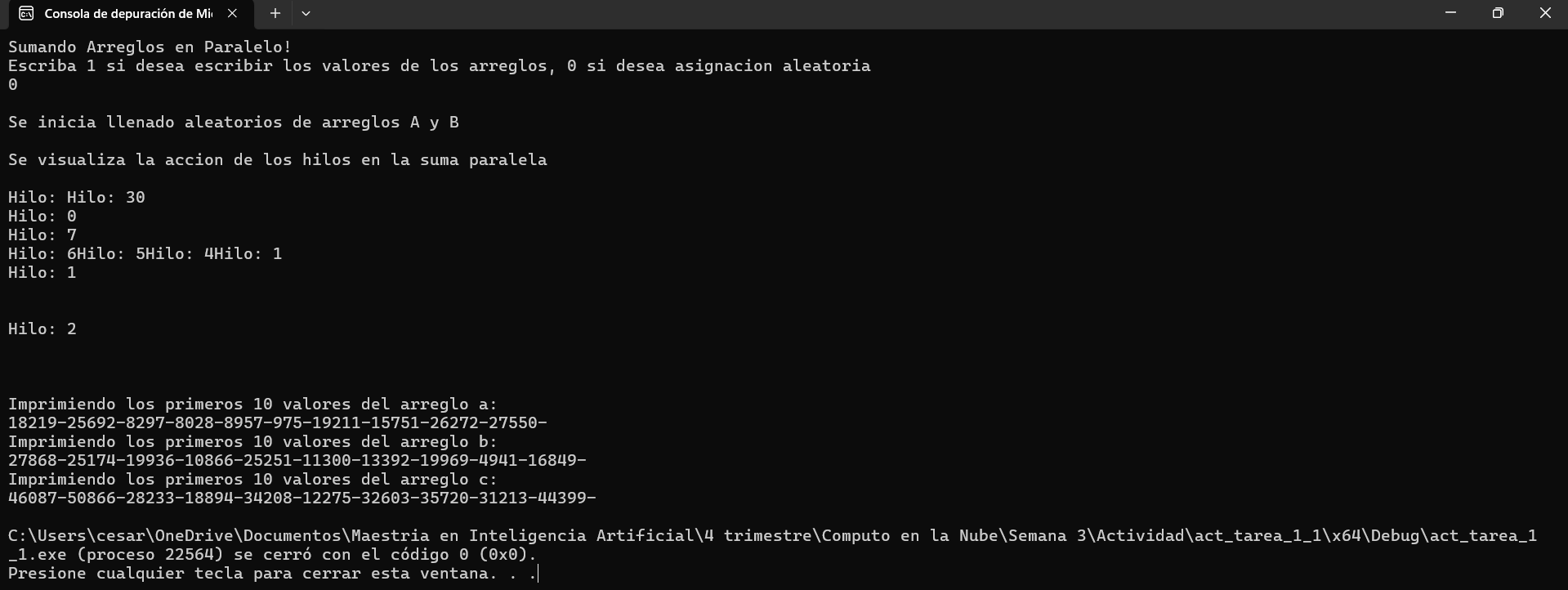
La clave para poder aprovechar al máximo la programación paralela consiste en identificar adecuadamente cuáles tareas pueden ser resultas bajo este esquema, pero, sobre todo, entender la manera en la que se pueda implementar.

Una de las prácticas más aceptadas dentro de la programación paralela es separar la tarea en tareas pequeñas que puedan ejecutarse de manera concurrente e independiente una de otra, esto a través de procesos que se conocen como hilos.

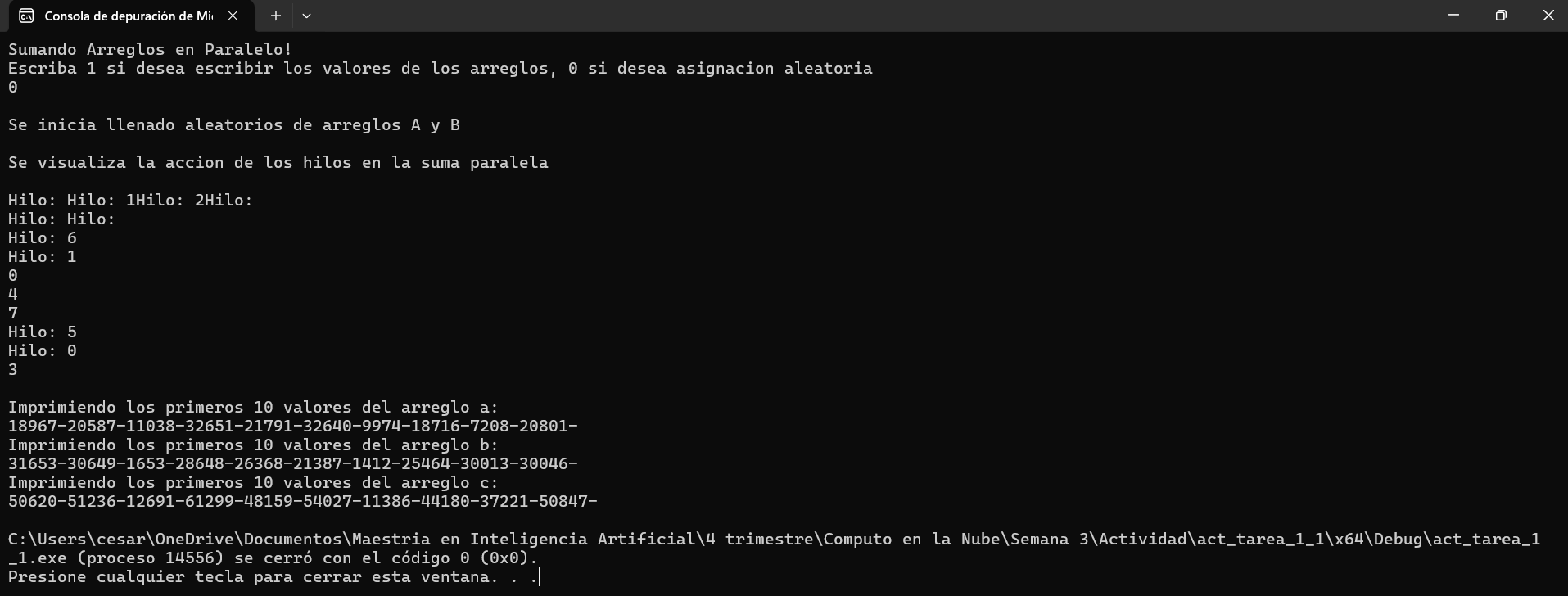
En la siguiente actividad se explorará la implementación de la programación paralela a través de hilos utilizando la librería OpenMP en el lenguaje de programación C, así como se analizará cómo los hilos trabajan y nos permiten llevar un proceso de manera eficiente al ejecutarse de forma paralela.

**Pantallas de ejecuciones**

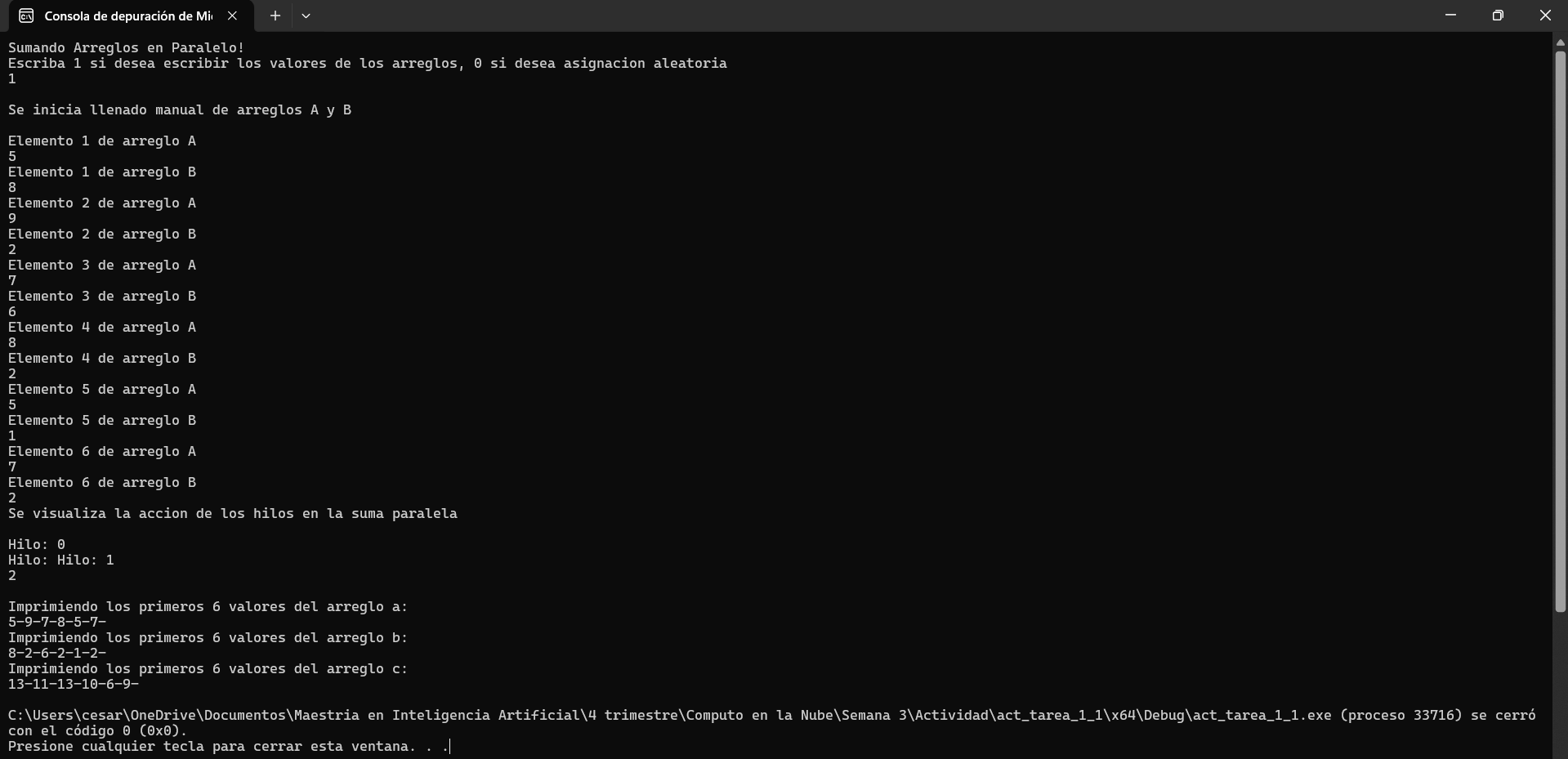
Pantalla de primera ejecución en donde se solicita el llenado de los arreglos de manera aleatoria



Pantalla de segunda ejecución en donde se solicita el llenado de los arreglos de manera aleatoria



Pantalla de ejecución en la que se solicita el llenado manual de los arreglos



**Explicación del código**

El objetivo general del código es la generación de un arreglo el cual se cree a partir de la suma paralela de otros dos arreglos cuyos valores sean proporcionados por el usuario o sean generados de manera automáticamente, ya sea la opción que prefiera el usuario.

El alcance general de la programación paralela en esta práctica es dividir nuestros arreglos en partes iguales y cada una de esas partes sea llevada por un hilo, el cual maneje su correspondiente parte y sume cada uno de sus elementos. De esta forma garantizamos llevar el proceso de suma de los arreglos a una eficiencia superior al “partir” la tarea en tareas más pequeñas.

Iniciamos la explicación del código:

NOTA: A continuación, se anexarán partes del código y los números en gris y con asteriscos serán los puntos de referencia que se explicarán después del código

#define N 1000 \*\*1\*\*

#define chunk 100 \*\*2\*\*

#define mostrar 10 \*\*3\*\*

\*\*1\*\*: Definimos la cantidad de elementos que tendrá cada arreglo, y por consecuente, la cantidad de elementos que tendrá el arreglo resultante

\*\*2\*\*: Definimos la cantidad de elementos que tomará cada subtarea en un hilo, es decir, se tomarán 100 elementos para llevar sumas independientes

\*\*3\*\*: Definimos la cantidad de elementos que queremos mostrar de cada arreglo

Hasta esta parte hemos definido el largo de los arreglos, el número de elementos para cada hilo y la cantidad de elementos a imprimir en la consola

int main() \*\*4\*\*

{

std::cout << "Sumando Arreglos en Paralelo! \n";

float a[N], b[N], c[N] \*\*5\*\*;

int i;

int respuesta;

std::cout << "Escriba 1 si desea escribir los valores de los arreglos, 0 si desea asignacion aleatoria\n"

cin >> respuesta \*\*6\*\*;

std::cout << "\n";

\*\*4\*\*: Construimos la función general para el llenado de los arreglos, así como la creación del tercer arreglo

\*\*5\*\*: Definimos nuestros tres arreglos cuyo largo va en función al número de elementos previamente definidos

\*\*6\*\*: Previo a la definición de esta variable hemos preguntado al usuario si desea llenar manualmente cada uno de los elementos de los arreglos A y B, o si bien, si desea que automáticamente se llenen dichos arreglos con números aleatorios. Si desea el llenado manual debe teclear 1; 0 en caso contrario.

Hasta esta parte sabremos la manera en la que se llenarán los arreglos.

std::cout << "\n";

srand(time(NULL)) \*\*7\*\*;

for (i = 0;i < N;i++)

{

int num\_a;

int num\_b;

if (respuesta == 1) \*\*8\*\*

{

std::cout << "Elemento " << i+1 << " de arreglo A"<< std::endl;

cin >> num\_a\*\*9\*\*;

std::cout << "Elemento " << i+1 << " de arreglo B" << std::endl;

cin >> num\_b \*\*10\*\*;

a[i] = num\_a;

b[i] = num\_b;

}

else

{

a[i] = rand() \*\*11\*\*;

b[i] = rand();

}

}

\*\*7\*\*: Haciendo uso de la función *srand* generamos números aleatorios a través de la función *time*, esto para procurar que cada vez que se ejecute el programa tengamos números aleatorios diferentes, debido a que el tiempo será diferente en cada ejecución

\*\*8\*\*: Basado en la respuesta que ha dado el usuario sobre el llenado de los arreglos se especifica la manera en la que se construirán los arreglos

\*\*9\*\*: Para el caso del llenado manual se solicitará cada elemento, empezando con el arreglo A

\*\*10\*\*: De la misma forma que el arreglo A se va llenando el arreglo B

\*\*11\*\*: Para el caso del llenado automático por valores aleatorios usaremos la función *rand()*, esto para ambos arreglos

Hasta este punto ya hemos terminado de llenar los arreglos

int pedazos = chunk\*\*12\*\*;

std::cout << "Se visualiza la accion de los hilos en la suma paralela \n";

std::cout << "\n";

#pragma omp parallel for \ \*\*13\*\*

shared(a,b,c,pedazos) private(i) \ \*\*14\*\*

schedule(static,pedazos) \*\*15\*\*

for (i = 0; i < N; i++) {

/\*Ponemos una sentencia para ver la accion de los hilos\*/

/\*Con la siguiente condicion vemos si es el comienzo de un nuevo pedazo\*/

if (i % pedazos == 0) \*\*16\*\*

{

int thread\_id = omp\_get\_thread\_num() \*\*17\*\*;

std::cout << "Hilo: " << thread\_id <<std::endl;

}

c[i] = a[i] + b[i] \*\*18\*\*;

}

\*\*12\*\*: La variable *pedazos* especificará el número de elementos que se considerará en cada hilo para su correspondiente proceso de suma

\*\*13\*\*: Utilizando la función *OpenMP* especificamos que el ciclo *for* (ciclo donde se llevará la creación del arreglo C a partir de los dos arreglos) se hará de forma paralela por los distintos hilos

\*\*14\*\*: Las variables *a, b, c* y *pedazos* se compartirán entre todos los hilos, en contraste a la variable *i*, la cual se necesita que quede privada para cada hilo y así mantenerse independiente entre cada uno.

\*\*15\*\*: Se lleva la planificación como estática, es decir, de manera uniforme se llevan los hilos

\*\*16\*\*: Bajo esta condición se revisa si es el inicio del bloque de 100 iteraciones de cada hilo, esto obteniendo el resultado de la división de *i* y *pedazos*. Si el resultado de esta operación da 0 es porque el valor de *i* es múltiplo de 100, es decir, es el inicio de cada bloque de 100 iteraciones.

\*\*17\*\*: En caso de cumplirse la condición de inicio de hilo imprimimos su correspondiente índice

\*\*18\*\*: Entonces, de manera paralela, se irá construyendo el arreglo C a partir de la ejecución de cada hilo, guardando cada elemento de los 100 que le corresponde a cada subtarea. La variable *i* determina la posición que corresponde llevar para cada elemento, por lo que ningún hilo va a interferir en la ejecución de algún otro hilo al llenar elementos totalmente independientes.

Hasta esta parte se ha construido el tercer arreglo

std::cout << "Imprimiendo los primeros " << mostrar << " valores del arreglo a: " << std::endl\*\*19\*\*;

imprimeArreglo(a);

std::cout << "Imprimiendo los primeros " << mostrar << " valores del arreglo b: " << std::endl;

imprimeArreglo(b);

std::cout << "Imprimiendo los primeros " << mostrar << " valores del arreglo c: " << std::endl;

imprimeArreglo(c);

}

void imprimeArreglo(float\* d)

{

for (int x = 0; x < mostrar;x++) \*\*20\*\*

std::cout << d[x] << "-";

std::cout << std::endl;

}

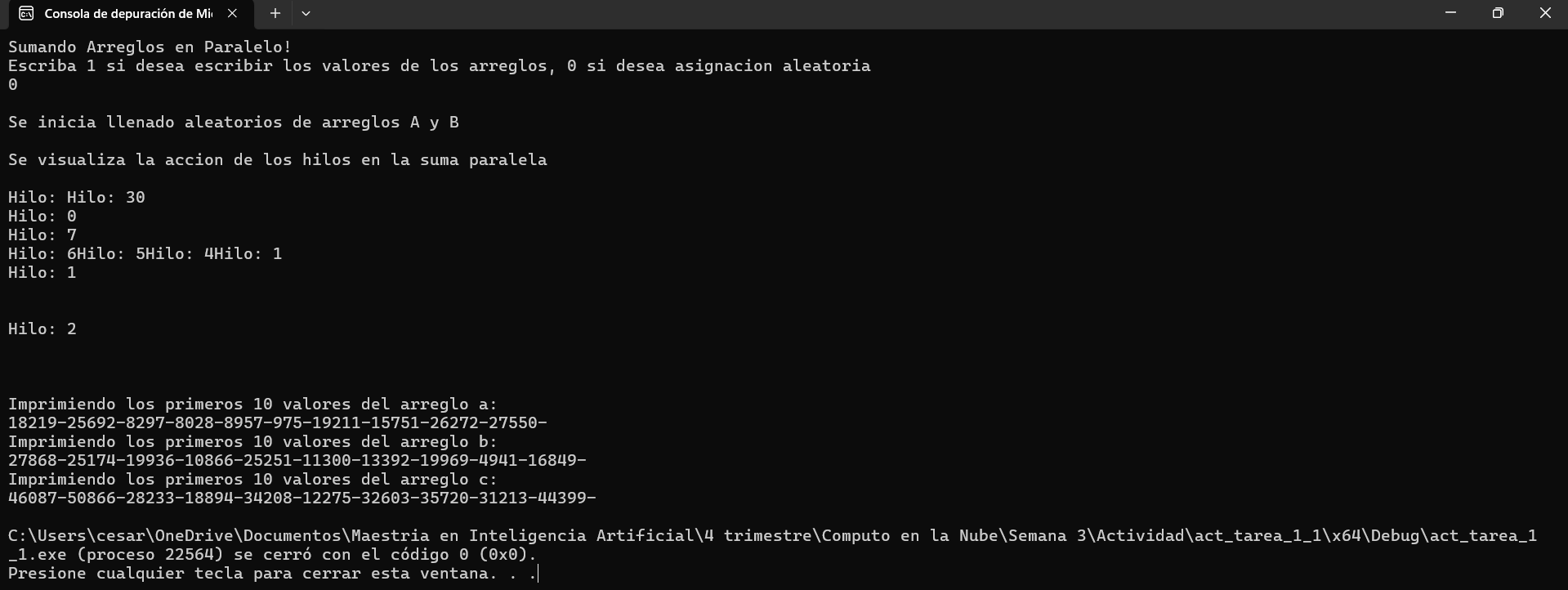
\*\*19\*\*: Basándonos en la variable *mostrar* se define cuántos elementos vamos a mostrar, lo que nos permitirá visualizar si el arreglo C fue construido correctamente

\*\*20\*\*: Esta impresión es llevada por la función *imprimeArreglo*, la cual es llevada por un ciclo en el que la variable *mostrar* indica hasta cuándo se ejecutará.

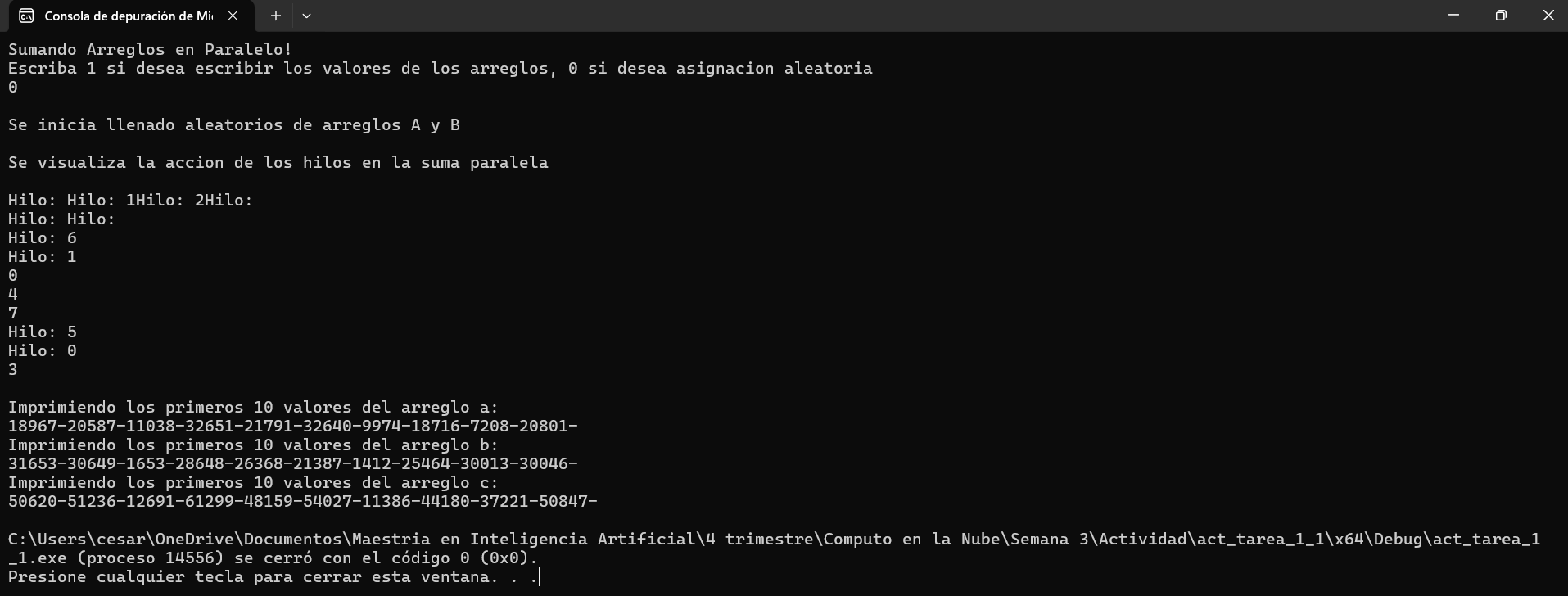
Esta función es usada para la impresión de los tres arreglos, cada uno es usado como argumento dentro de la función.

**Explicación de resultados**

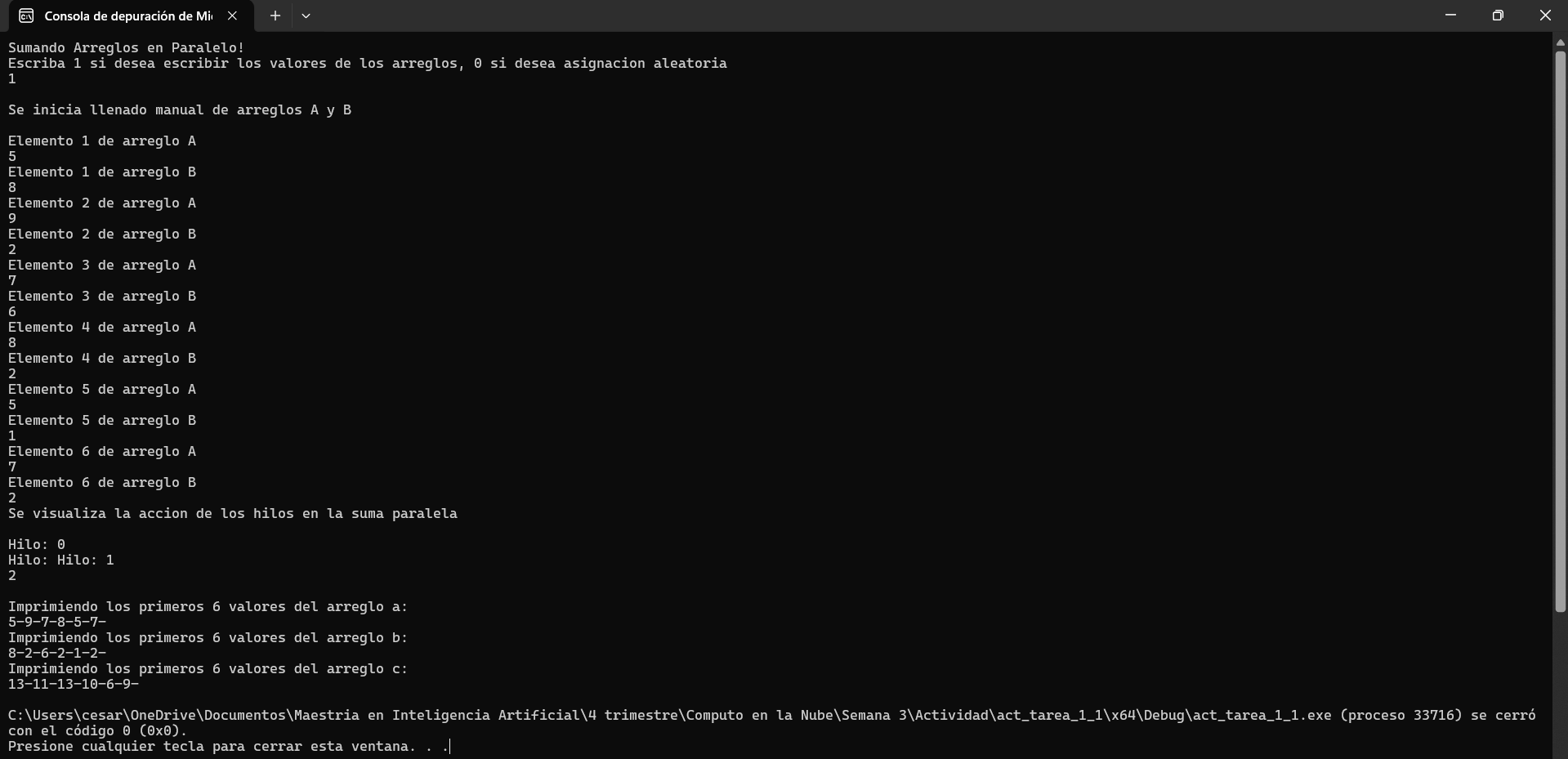
En esta sección se explicarán los resultados de las 3 pantallas proporcionadas previamente.



En esta primera ejecución hemos elegido la opción de llenado de arreglos A y B de manera automática con valores aleatorios. Asimismo, podemos observar cómo los hilos se ejecutaron de manera paralela y asíncrona, lo que hace que se traslapen sus respectivas impresiones. Al final vemos cómo el arreglo C fue construido correctamente.



En esta ejecución igualmente consideramos la opción de llenado por valores aleatorios. Lo relevante de esta ejecución es que podemos ver que los elementos de los arreglos A y B son distintos a los elementos de la pasada ejecución, lo que nos confirma que la utilización de la función *srand(time(NULL))* nos permitió tener valores distintos para cada ejecución.



Para esta última ejecución hemos elegido la opción de llenado manual de arreglos, por lo que, para mayor practicidad, hemos modificado los valores de las variables *N, chunk* y *mostrar.* Los nuevos valores son 6, 2 y 6. Estas modificaciones igualmente han permitido ver cómo la cantidad de hilos se reduce solo a 3.

**Reflexiones**

Sin duda alguna la programación paralela nos puede ayudar en tareas que normalmente requieren mucho procesamiento, conllevando así mayores tiempos de ejecución, esto aprovechando los distintos procesadores que nuestras computadoras tienen.

Considero que la clave del éxito en esta práctica es saber exactamente cuáles tareas podrían tener beneficios si aplicamos la programación paralela, certificando que se pueda llevar por distintas subtareas. Pero, sobre todo, considero que la más importante parte de garantizar el éxito es entender cómo funciona nuestro programa y cómo podemos configurar sus distintas partes y segmentos para su correspondiente división en hilos.

Es decir, considero que es muy relevante entender cuáles puntos del código se modificarán y la manera en la que se modificarán para que puedan ser llevados por tareas que se ejecuten de manera paralela. Asimismo, es de especial valor saber cómo los hilos se detonan, para así validar que estos subprocesos no se interpongan entre ellos y generen errores de ejecución, o proporcionen otros resultados a los esperados.