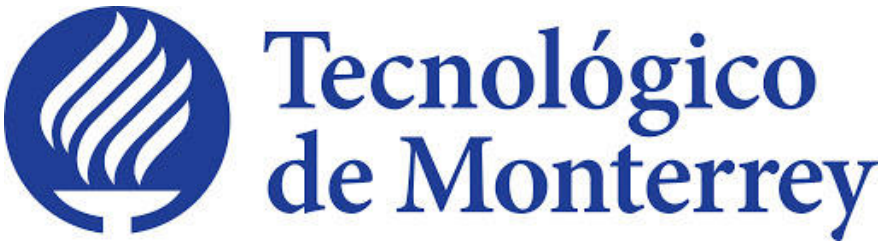

Instituto Tecnológico y de Estudios Superiores de Monterrey

Maestría en Inteligencia Artificial:

Tarea1: Programación de una solución paralela

Alumno: Oscar Enrique García García - A01016093

Profesor Titular: Gilberto Echeverría Furió

Profesor asistente: Yetnalezi Quintas Ruiz

Cómputo en la nube

Enero, 2026

Índice

1. Introducción

Tradicionalmente, el software se ha escrito para el cómputo secuencial: un programa recibe una instrucción, la ejecuta y, solo cuando termina, pasa a la siguiente. Sin embargo, los procesadores modernos cuentan con múltiples núcleos (cores) que pueden trabajar simultáneamente.

La programación en paralelo es el proceso de dividir un problema grande en partes más pequeñas que pueden resolverse al mismo tiempo, aprovechando cada núcleo del procesador de forma concurrente y, de esta forma, optimizar el tiempo de ejecución.

El uso de paralelismo es fundamental en el desarrollo actual por varias razones:

- Reducción de Tiempo: Tareas que tardarían horas en un solo hilo pueden completarse en minutos al distribuir la carga.
- Manejo de Big Data: Permite procesar volúmenes masivos de información que, de otro modo, saturarían un solo núcleo.
- Eficiencia Energética: A menudo es más eficiente realizar un trabajo rápido usando varios núcleos a una frecuencia moderada que forzar un solo núcleo a su máxima capacidad por mucho tiempo.

En este proyecto, utilizamos OpenMP (Open Multi-Processing), una interfaz de programación de aplicaciones (API) que permite añadir paralelismo a códigos escritos en C++ mediante «pragmas» o directivas de compilador. Es una de las herramientas más potentes para cómputo de alto rendimiento debido a su facilidad de implementación en sistemas de memoria compartida.

El código fuente del proyecto se encuentra disponible en el **repositorio de GitHub:**

https://github.com/oscargarciatec/Cloud_Computing/blob/main/Tarea1_Parallel_Computing/Tarea1.cpp

2. Capturas de pantalla de los ejercicios realizados

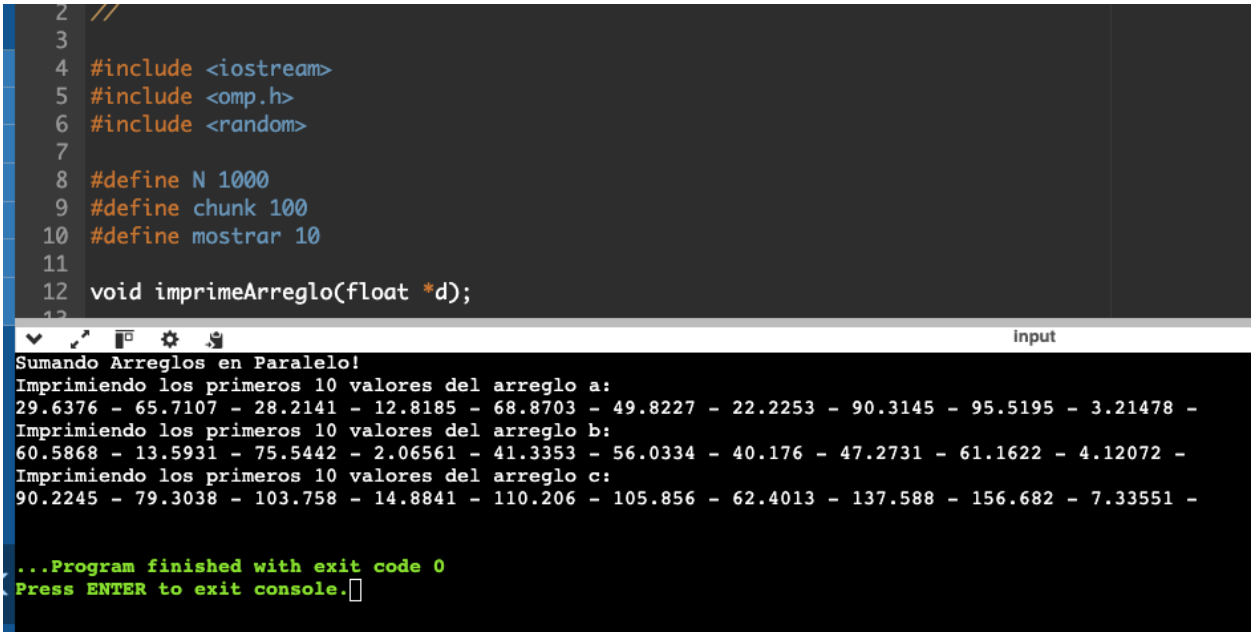

Figura 1: Ejecución del proyecto con 1000 números generados, chunks de tamaño 100 e impresión de 10 resultados.

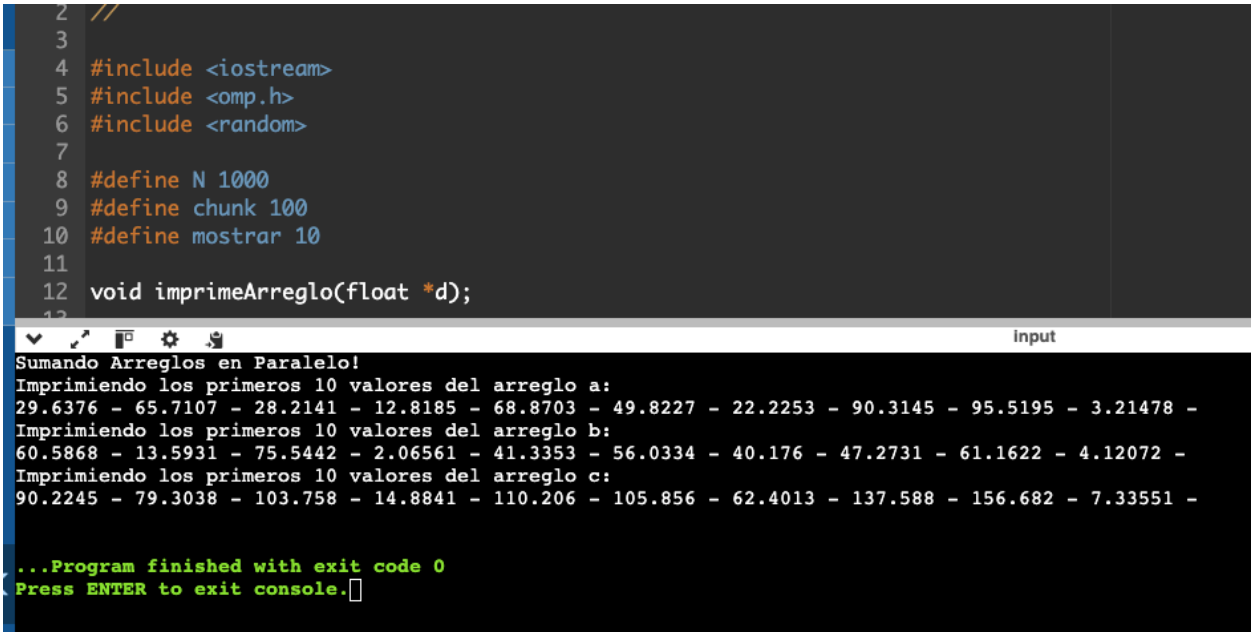

Figura 2: Ejecución del proyecto con 100000 números generados, chunks de tamaño 1000 e impresión de 100 resultados.

3. Explicación del código y los resultados

El código se divide en 4 secciones principales:

1. Importación de librerías, definición de variables y definición de funciones.
2. Generación de números aleatorios para los arreglos a y b.
3. Generación del arreglo c con la suma de los números generados en el paso anterior.
4. Impresión de resultados.

3.1. Importación de librerías, definición de variables y definición de funciones.

Estas primeras líneas del código se centran en la importación de las librerías que usamos para todo el código, incluyendo la librería *random* que nos servirá para generar los números aleatorios.

```
#include <iostream>
#include <omp.h>
#include <random>
```

A continuación, se definen las variables «N» que es el tamaño de cada uno de los arreglos (números que se generarán), «chunk» que corresponde al tamaño de esos «pedazos» en los que se divide la tarea entre los hilos y «mostrar» que hace referencia a cuántos números se mostrarán en la impresión de resultados. De igual forma, se declara nuestra función «imprimeArreglo» que recibirá un arreglo de números decimales como parámetro.

```
#define N 100000
#define chunk 1000
#define mostrar 100

void imprimeArreglo(float *d);
```

Finalmente, se define el código de la función imprimeArreglo que básicamente recorre el arreglo de números flotantes que recibe como parámetro e imprime cada «posición» (cada número), separada por guiones medios.

```
void imprimeArreglo(float* d)
{
  for (int x = 0; x < mostrar; x++)
    std::cout << d[x] << " - ";
  std::cout << std::endl;
}
```

3.2. Generación de números aleatorios para los arreglos a y b.

En el siguiente bloque se inicializa la función main del código y se crean los arreglos a, b y c, de tamaño N (la variable que declaramos al inicio). De igual forma, se define la variable «i» que nos servirá como apuntador en nuestro ciclo for que genera los números aleatorios para cada arreglo y finalmente, se declara la variable local «pedazos» que será del mismo valor que la variable «chunk» del inicio.

```
int main()
{
  std::cout << "Sumando Arreglos en Paralelo!\n";
  float a[N], b[N], c[N];
  int i;
  int pedazos = chunk;
```

Este bloque de código es el encargado de generar los números aleatorios. En este bloque hay un par de cosas que vale la pena recalcar:

1. Se utiliza pragma omp parallel con el fin de repartir las tareas del bloque en diferentes hilos.
2. La variable «i» se define como privada para evitar «colisiones» o sobreescritura de la variable entre los diferentes hilos.

3. Se emplea el algoritmo mt19937 con una semilla única por hilo (tid) para asegurar que la generación de números sea independiente. Adicionalmente, se utiliza uniform_real_distribution para generar números entre 1 y 100.
4. Se utiliza pragma omp for schedule(static, pedazos) con el fin de repartir las n iteraciones del ciclo for entre todos los hilos disponibles. Con esto, finalmente se guarda cada número generado en cada una de las posiciones de los arreglos a y b.

```
#pragma omp parallel private(i)
  {
    int tid = omp_get_thread_num();
    std::mt19937 gen(std::random_device{}() ^ tid);
    std::uniform_real_distribution<float> dis(1.0, 100.0);

    #pragma omp for schedule(static, pedazos)
    for (i = 0; i < N; i++)
    {
      a[i] = dis(gen);
      b[i] = dis(gen);
    }
  }
```

3.3. Generación del arreglo c

Una vez generados los arreglos a y b, se crea un nuevo ciclo for que, de igual forma, repartirá las n iteraciones del mismo entre todos los hilos disponibles.

Como podemos ver, el ciclo for calcula la suma entre el número en la posición «i» del arreglo «a» con el número en la misma posición del arreglo «b» y guarda el resultado en el arreglo «c», para cada una de las posiciones en los arreglos.

```
#pragma omp parallel for \
  shared(a,b,c, pedazos) private(i) \
  schedule(static, pedazos)
```

```
  for (i = 0; i < N; i++)
    c[i] = a[i] + b[i];
```

3.4. Impresión de resultados

Finalmente, se hace una impresión de resultados, utilizando la función imprimeArreglo para imprimir lor primeros n resultados, definidos por la variable «mostrar», para cada uno de los arreglos.

De esta forma, podemos verificar que los resultados son los correctos; que, efectivamente, cada posición en «c» es el resultado de la suma de los números en la misma posición de los arreglos «a» y «b».

```
std::cout << "Imprimiendo los primeros " << mostrar << " valores del arreglo a: "
<< std::endl;
  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);
}
```

3.5. Código Completo

```
#include <iostream>
#include <omp.h>
#include <random>

#define N 100000
#define chunk 1000
#define mostrar 100

void imprimeArreglo(float *d);

void imprimeArreglo(float* d)
{
  for (int x = 0; x < mostrar; x++)
    std::cout << d[x] << " - ";
  std::cout << std::endl;
}

int main()
{
  std::cout << "Sumando Arreglos en Paralelo!\n";
  float a[N], b[N], c[N];
  int i;
  int pedazos = chunk;

  #pragma omp parallel private(i)
  {
    int tid = omp_get_thread_num();
    std::mt19937 gen(std::random_device{}() ^ tid);
    std::uniform_real_distribution<float> dis(1.0, 100.0);

    #pragma omp for schedule(static, pedazos)
```

```
    for (i = 0; i < N; i++)
    {
      a[i] = dis(gen);
      b[i] = dis(gen);
    }
  }

  #pragma omp parallel for \
  shared(a,b,c, pedazos) private(i) \
  schedule(static, pedazos)

  for (i = 0; i < N; i++)
    c[i] = a[i] + b[i];

  std::cout << "Imprimiendo los primeros " << mostrar << " valores del arreglo a:
" << std::endl;
  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);
}
```

4. Reflexión

La programación paralela, ejemplificada en este proyecto con OpenMP, representa un cambio de paradigma respecto a la ejecución secuencial tradicional. De este ejercicio se desprenden las siguientes conclusiones:

- **Eficiencia en Escala**: Para un arreglo de 1,000 elementos, la diferencia de tiempo es imperceptible, pero cuando escalamos a millones de datos o simulaciones físicas complejas, la capacidad de dividir el «bucle for» entre los núcleos disponibles del CPU reduce el tiempo de ejecución casi de forma lineal.
- **Gestión de Memoria**: El código utiliza la cláusula private(i) y shared(a,b,c). Esto es crítico: si el índice «i» no fuera privado, los hilos entrarían en una «condición de carrera» (race condition), intentando modificar la misma variable de control simultáneamente, lo que colapsaría el programa.
- **Abstracción de Complejidad**: OpenMP permite que el desarrollador se concentre en la lógica del algoritmo mientras la librería se encarga de la creación, sincronización y destrucción de los hilos (threads).

Finalmente, el ejercicio demuestra que la programación paralela no es solo una técnica de optimización, sino una necesidad arquitectónica. Al dominar herramientas como OpenMP, logramos que el software sea capaz de aprovechar el hardware multincúcleo actual, superando los límites físicos de la velocidad de procesamiento de un solo núcleo y abriendo la puerta a procesamientos de datos masivos con un control preciso sobre la memoria y la sincronización.

5. Referencias

[1] D. B. Kirk y W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands--on Approach*, 3rd ed. Burlington, MA, USA: Morgan Kaufmann, 2016. [En línea]. Disponible en: https://learning.oreilly.com/library/view/programming-massively--parallel/9780128119877/

[2] R. Robey y Y. Zamora, *Parallel and High Performance Computing*. Shelter Island, NY, USA: Manning Publications, 2021. [En línea]. Disponible en: https://learning.oreilly.com/library/view/parallel-and-high/9781617296468/

[3] OpenMP, «OpenMP Reference Guides». [En línea]. Disponible en: https://www.openmp.org/resources/refguides/

[4] Matsson, Tim, «A "Hands-on" Introduction to OpenMP». [En línea]. Disponible en: https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG