Tecnológico de Monterrey

Cómputo en la Nube (Gpo 10)

Tarea 1. Programación de una solución paralela

Estudiante:

Arango Restrepo, Isabella - A01796305

Profesores:

Julio César Salgado

Gilberto Echeverría Furió

29 de enero de 2026

Introducción

La programación paralela representa una alternativa importante frente a los enfoques secuenciales tradicionales, especialmente cuando se trabaja con problemas que involucran grandes volúmenes de datos. En el curso, se ha mencionado que, aunque los procesadores actuales cuentan con múltiples núcleos, no siempre se aprovechan de manera eficiente si los algoritmos se diseñan de forma estrictamente secuencial. Desde esta perspectiva, el paralelismo se presenta como una herramienta clave para mejorar el rendimiento y comprender mejor cómo utilizar los recursos computacionales disponibles.

En esta actividad se aborda la implementación de una solución paralela para la suma de dos arreglos numéricos, aprovechando que las operaciones entre los elementos son independientes y pueden ejecutarse de manera simultánea. Aunque se trata de un problema sencillo, resulta especialmente útil para afianzar los conceptos básicos de programación paralela y para observar, de manera práctica, cómo una tarea puede dividirse y distribuirse entre múltiples hilos de ejecución.

La solución se desarrolla utilizando la librería **OpenMP** [1], haciendo uso de un ciclo for paralelo que permite repartir las iteraciones entre los hilos disponibles. A través de esta implementación, es posible comparar el enfoque paralelo con una solución secuencial y reflexionar sobre las ventajas, limitaciones y consideraciones que implica el uso del paralelismo. El desarrollo y ejecución del programa se realizó mediante la plataforma **OnlineGDB**, lo cual permitió enfocarse en la lógica del paralelismo y en la comprensión del código sin necesidad de realizar instalaciones locales.

Explicación del código

La tarea fue desarrollada utilizando el lenguaje **C++** en la plataforma **OnlineGDB**, dentro de un proyecto denominado **proyectosOMP**, tratando de emular la creación de proyectos en Visual Studio.

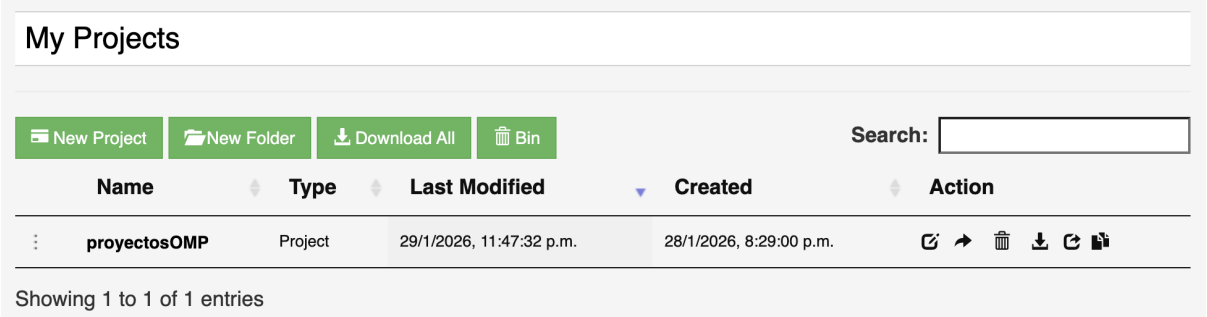

Figura 1. Proyecto en OnlineGDB.

Esta herramienta permitió compilar y ejecutar el código sin necesidad de realizar instalaciones locales, además de contar con soporte para la librería **OpenMP**, necesaria para la implementación del paralelismo. Para su correcto funcionamiento, es necesario la configuración de las flags tal como se muestra a continuación:

Figura 2. Configuración de glags de compilación

El código fuente y los archivos asociados al proyecto fueron almacenados y versionados en el repositorio de GitHub indicado:

Repositorio Github: https://github.com/isabellaarangor/computo-en-la-nube-2026/tree/main

En las próximas líneas se detallan cada una de las partes que componen el código utilizado.

En primer lugar, se incluyen las librerías necesarias para el funcionamiento del programa. La librería *<iostream>* se utiliza para la entrada y salida de información en consola, mientras que *<omp.h>* permite el uso de OpenMP, las cuales hacen posible la ejecución paralela del código.

```
1 #include <iostream>
2 #include <omp.h>
```

Figura 3. Importación de librerías

Posteriormente, se definen tres constantes de precompilación:

- **N**, que indica el número total de elementos que tendrán los arreglos.
- **chunk**, que especifica el tamaño de los bloques de iteración que serán asignados a cada hilo durante la ejecución paralela.
- **mostrar**, que determina cuántos elementos de cada arreglo se imprimirán en pantalla para verificar los resultados.

El uso de estas constantes facilita la modificación y experimentación con diferentes tamaños de datos sin alterar la lógica principal del programa.

```
4 #define N 600000
5 #define chunk 100
6 #define mostrar 10
```

Figura 4. Definición de constantes

Antes de la función principal, se declara el prototipo de la función **imprimeArreglo**, la cual recibe como parámetro un apuntador a un arreglo de tipo float. Esta función se encarga de imprimir únicamente los primeros elementos definidos por la constante **mostrar**, lo que permite validar el contenido de los arreglos sin saturar la salida en consola.

```
void imprimeArreglo(float *d);
```

Figura 5. Declaración de función.

Dentro de la función principal **main**, el programa inicia mostrando un mensaje en consola para indicar el comienzo de la ejecución. Posteriormente, se realiza una validación del funcionamiento de las directivas de **OpenMP** mediante el uso de la directiva de preprocesador **_OPENMP**.

Para esta verificación, se utiliza una instrucción condicional, la cual permite comprobar si el código fue compilado con soporte para OpenMP habilitado. En caso afirmativo, se imprime el mensaje “OMP disponible y funcionando”, confirmando que el entorno de ejecución reconoce correctamente las directivas de paralelismo. Esta validación es importante, ya que permite identificar posibles problemas de configuración; si OpenMP no estuviera activo, el programa se ejecutaría de manera secuencial.

```
10  int main()
11  {
12      std::cout << "Verificando la ejecución de las directivas OMP!\n";
13
14      #ifdef _OPENMP
15          std::cout << "OMP disponible y funcionando\n" << std::endl;
```

Figura 6. Validación del funcionamiento de OpenMP.

Una vez confirmado el funcionamiento de OpenMP, se procede a declarar los arreglos a, b y c, todos de tamaño N. Los arreglos a y b se utilizan para almacenar los valores iniciales, mientras que el arreglo c se reserva para guardar el resultado de la suma elemento a elemento de los dos arreglos anteriores.

A continuación, se inicializan los arreglos a y b mediante un ciclo for secuencial, asignando valores calculados a cada una de sus posiciones. Esta inicialización se realiza de forma secuencial debido a que su costo computacional es bajo y no representa un impacto significativo en el desempeño del programa.

```
17          std::cout << "Sumando Arreglos en Paralelo!\n";
18          float a[N], b[N], c[N];
19          int i;
20
21          for (i = 0; i < N; i++)
22          {
23              a[i] = i * 10;
24              b[i] = (i + 3) * 3.7;
25          }
```

Figura 7. Inicialización de arreglos.

La parte central del programa, y la principal para el cumplimiento del objetivo de la tarea, corresponde a la suma paralela de los arreglos. Para ello, se utiliza la directiva #pragma omp parallel for, la cual permite distribuir las iteraciones del ciclo for entre múltiples hilos de ejecución.

En esta directiva se especifica que los arreglos a, b y c, así como la variable que define el tamaño del bloque (pedazos), serán compartidos entre los hilos. Por otro lado, la variable de control del ciclo (i) se declara como privada para que cada hilo tenga su propio valor y así evitar que se presenten errores por interferencia entre hilos. Además, se emplea la cláusula schedule(static, pedazos), la cual indica que las iteraciones se repartirán en bloques de tamaño fijo entre los hilos disponibles, permitiendo un reparto predecible del trabajo.

Cada hilo realiza la suma de un subconjunto independiente de los arreglos, almacenando el resultado directamente en el arreglo c. Dado que cada posición del arreglo es escrita por una única iteración, no se presentan conflictos de acceso a memoria.

```
27          int pedazos = chunk;
28
29          #pragma omp parallel for \
30              shared(a, b, c, pedazos) private(i) \
31              schedule(static, pedazos)
32          for (i = 0; i < N; i++)
33              c[i] = a[i] + b[i];
```

Figura 8. Paralelización de la suma de arreglos.

Una vez finalizada la ejecución paralela, se imprimen los primeros elementos de los arreglos a, b y c utilizando la función **imprimeArreglo**. Esta salida permite comprobar que la suma se realizó correctamente y que los valores del arreglo resultante corresponden a la suma elemento a elemento de los arreglos originales.

```
35          std::cout << "Imprimiendo los primeros " << mostrar
36                    << " valores del arreglo a: " << std::endl;
37          imprimeArreglo(a);
38
39          std::cout << "Imprimiendo los primeros " << mostrar
40                    << " valores del arreglo b: " << std::endl;
41          imprimeArreglo(b);
42
43          std::cout << "Imprimiendo los primeros " << mostrar
44                    << " valores del arreglo c: " << std::endl;
45          imprimeArreglo(c);
```

Figura 9. Muestra de resultados.

Finalmente, se define la función **imprimeArreglo**, la cual recorre las primeras posiciones del arreglo recibido como parámetro y muestra sus valores en consola. Esta función se reutiliza para imprimir los tres arreglos, promoviendo la modularidad y claridad del código.

```
52  void imprimeArreglo(float *d)
53  {
54      for (int x = 0; x < mostrar; x++)
55          std::cout << d[x] << " - ";
56      std::cout << std::endl;
57  }
```

Figura 10. Definición de la función para imprimir los arreglos.

Finalmente, el código completo del programa, incluyendo la validación de OpenMP, la inicialización de los arreglos, la suma paralela y la impresión de resultados, se muestra en la figura 11. Este código integra todas las secciones explicadas previamente y permite observar de manera clara cómo se implementa una solución paralela para la suma de arreglos utilizando la librería OpenMP.

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

#define N 600000
#define chunk 100
#define mostrar 10

void imprimeArreglo(float *d);

int main()
{
    std::cout << "Verificando la ejecución de las directivas OMP!\n";

    #ifdef _OPENMP
        std::cout << "OMP disponible y funcionando\n" << std::endl;

        std::cout << "Sumando Arreglos en Paralelo!\n";
        float a[N], b[N], c[N];
        int i;

        for (i = 0; i < N; i++)
        {
            a[i] = i * 10;
            b[i] = (i + 3) * 3.7;
        }

        int pedazos = chunk;

        #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);

    #else
        std::cout << "OMP no disponible\n" << std::endl;
    #endif
}

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

Figura 11. Código completo utilizado.

Código en funcionamiento

Para validar el correcto funcionamiento del programa y el uso de paralelismo con OpenMP, se realizaron ejecuciones bajo configuraciones distintas. A continuación se detallada cada una de ellas:

Con el soporte de OpenMP habilitado

En la primera prueba, el programa se ejecutó utilizando el flag **-fopenmp**. Al hacer esto, el programa detecta que OpenMP está habilitado y muestra en pantalla el mensaje *“OMP disponible y funcionando”*, lo que confirma que el entorno soporta la ejecución en paralelo.

Después de esto, el programa realiza la suma de los arreglos a y b utilizando el ciclo en paralelo, es decir, repartiendo el trabajo entre los hilos disponibles. Al finalizar la ejecución, se imprimen los primeros diez valores de cada arreglo c con la suma correcta de los valores correspondientes de a y b.

```
OMP disponible y funcionando

Sumando Arreglos en Paralelo!
Imprimiendo los primeros 10 valores del arreglo a:
0 - 10 - 20 - 30 - 40 - 50 - 60 - 70 - 80 - 90 -
Imprimiendo los primeros 10 valores del arreglo b:
11.1 - 14.8 - 18.5 - 22.2 - 25.9 - 29.6 - 33.3 - 37 - 40.7 - 44.4 -
Imprimiendo los primeros 10 valores del arreglo c:
11.1 - 24.8 - 38.5 - 52.2 - 65.9 - 79.6 - 93.3 - 107 - 120.7 - 134.4 -
```

Figura 12. Resultado de la ejecución con OpenMP.

Sin soporte de OpenMP habilitado

En la segunda prueba, el programa se ejecutó sin utilizar el flag **-fopenmp**. En este caso, el programa muestra el mensaje *“OMP no disponible”*, indicando que las directivas de OpenMP no están activas.

Bajo esta configuración, el programa se ejecuta de manera secuencial, sin utilizar paralelismo. Aun así, la ejecución finaliza correctamente, lo que permite comprobar que el código funciona incluso sin OpenMP, aunque en este caso no se aprovechan las ventajas del procesamiento en paralelo.

```
Verificando la ejecución de las directivas OMP!
OMP no disponible
```

Figura 13. Resultado de la ejecución sin OpenMP.

Límite de OnlineGDB

En esta prueba se evaluó el comportamiento del programa modificando el valor del parámetro chunk con un N muy alto, en este caso el máximo valor con el que se pudo obtener resultados (**N = 698.400**).

En una primera ejecución, se utilizó un valor pequeño (chunk = 10), lo que provocó que el programa terminara con un error de segmentación (exit code 139) [2]. Este comportamiento se atribuye al alto consumo de memoria y a la sobrecarga generada por la creación de múltiples bloques pequeños en el ciclo paralelo, especialmente considerando el tamaño grande de los arreglos y las limitaciones del entorno de ejecución.

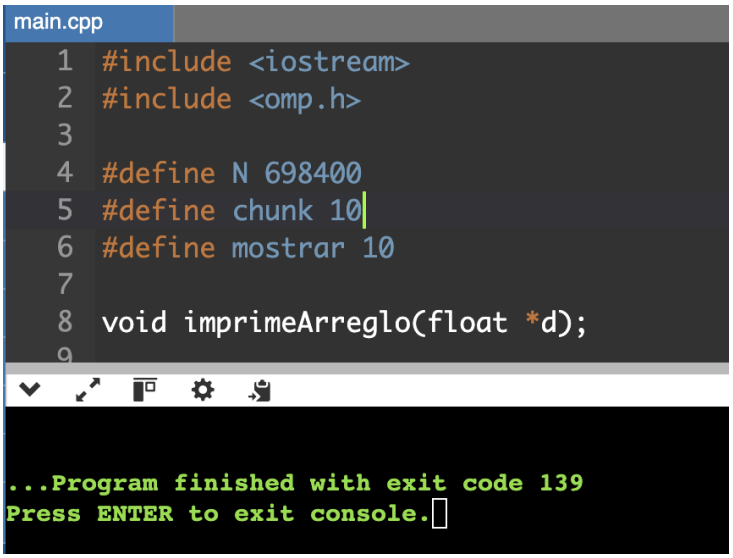

Figura 14. Resultado con exit code 139.

Posteriormente, se realizó una segunda prueba aumentando el valor de chunk a 10000. Bajo esta configuración, el programa se ejecutó correctamente, mostrando la validación de OpenMP y produciendo los resultados esperados en la suma de los arreglos. Este resultado indica que un tamaño de bloque mayor reduce la sobrecarga del paralelismo y mejora la estabilidad del programa.

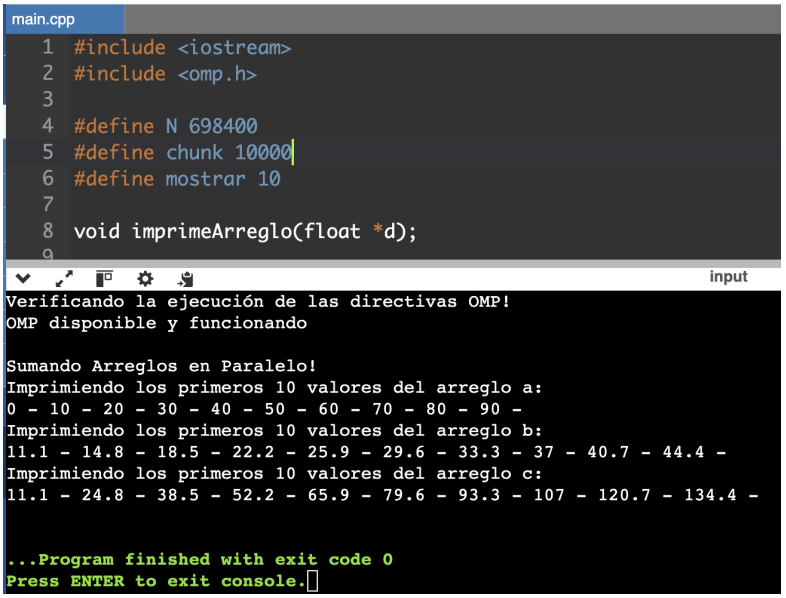

Figura 15. Resultado exitoso con chunk más alto.

Estas pruebas permiten observar que la correcta selección de parámetros en OpenMP es fundamental y que el desempeño y funcionamiento del paralelismo dependen tanto del algoritmo como del entorno en el que se ejecuta.

Reflexión

Durante el desarrollo de esta actividad fue posible entender que la programación paralela no se trata solo de agregar directivas para que un programa use varios hilos, sino de analizar bien el problema y el entorno donde se ejecuta. Aunque la suma de arreglos es un problema sencillo, permitió ver de manera práctica cómo se puede dividir una tarea y ejecutar varias partes al mismo tiempo.

Por otro lado, fue posible notar que el paralelismo no siempre funciona correctamente si no se eligen bien sus parámetros. En las pruebas realizadas, al usar un valor pequeño para el parámetro chunk junto con un tamaño grande de los arreglos, el programa presentó un error de segmentación. Esto ayudó a comprender que el uso de muchos bloques pequeños puede generar un mayor consumo de memoria y provocar errores, especialmente en un entorno con recursos limitados como OnlineGDB.

Al aumentar el valor de chunk, el programa se ejecutó correctamente y mostró los resultados esperados. Esto permitió observar que la forma en la que se reparte el trabajo entre los hilos influye directamente en el funcionamiento del programa. También quedó claro que crear y coordinar hilos tiene un costo, y que en algunos casos ese costo puede ser mayor que el beneficio de paralelizar una tarea.

En conclusión, esta práctica ayudó a entender que la programación paralela debe utilizarse de manera consciente. No siempre es la mejor opción para todos los problemas, pero cuando se aplica correctamente puede ser muy útil para trabajar con grandes cantidades de datos. Esta actividad permitió reforzar los conceptos vistos en clase y comprender mejor las ventajas y limitaciones del paralelismo.

Referencias

[1] OpenMP Architecture Review Board. (s. f.). About Us. OpenMP.org. Recuperado de https://www.openmp.org/about/about-us/

[2] GeeksforGeeks. (s. f.). *Exit codes in C/C++ with examples*. Recuperado el 31 de enero de 2026, de https://www.geeksforgeeks.org/cpp/exit-codes-in-c-c-with-examples/