

Curso Computo en la nube

Tarea 1: Programación de una solución paralela

Jesús Esteiner Alonso

Estudiante Máster en Inteligencia Artificial Aplicada

Universidad Tecnológico de Monterrey, Ciudad de México

A1793554@tec.mx

Enero 2023

Dr. Eduardo Antonio Cendejas Castro

Profesor titular MNA CN

Introducción

Debido a la reciente proliferación del uso compartido de memoria en las computadoras (*DSM distributed shared-memory*) por parte de la comunidad científica, hay demasiado interés en cómo utilizar de la mejor manera tanto la partición de memoria distribuida como la memoria compartida de estos sistemas computacionales. *MPI (message passing interface)* proporciona un medio eficiente de comunicación paralela entre una colección distribuida de máquinas de cómputo, sin embargo, no todas las implementaciones de MPI aprovechan la memoria compartida cuando está disponible entre los procesadores (la premisa básica es que dos procesadores, que comparten memoria común, pueden comunicarse entre si más rápido mediante el uso del medio compartido que a través de otros medios de comunicación). *OpenMP (open multi processing)* se introduce para proporcionar un medio para implementar el paralelismo de memoria compartida en programas como C/C++. Específicamente, OpenMP crea un conjunto de variables de entorno, directivas de compilación y rutinas que se utilizarán para la paralelización de memoria compartida. OpenMP se diseñó específicamente para explotar ciertas características de las arquitecturas de memoria compartida, como la capacidad de acceder directamente a la memoria en todo el sistema con baja latencia y bloqueos de memoria compartida de una manera muy rápida.

Está surgiendo un nuevo paradigma de programación paralela en el que tanto MPI como OpenMP se pueden utilizar para estos procesos de paralelización. En una arquitectura de memoria compartida distribuida DSM, OpenMP usa la comunicación entre nodos (es decir,

entre una colección de procesadores que comparten el mismo subsistema de memoria) y MPI se usaría para la comunicación entre los nodos (es decir, entre las distintas colecciones distribuidas de procesadores). La combinación de estas dos metodologías de paralelización puede proporcionar los medios más efectivos para explotar completamente los sistemas DSM modernos.

En esta oportunidad, estaremos implementando con la utilización de arreglos en C++, la sumatoria en paralelo de un conjunto de arreglos en donde podamos apreciar y aplicar la programación paralela sumando algunos elementos de los arreglos A y B, y generando un tercer arreglo C con el resultado obtenido.

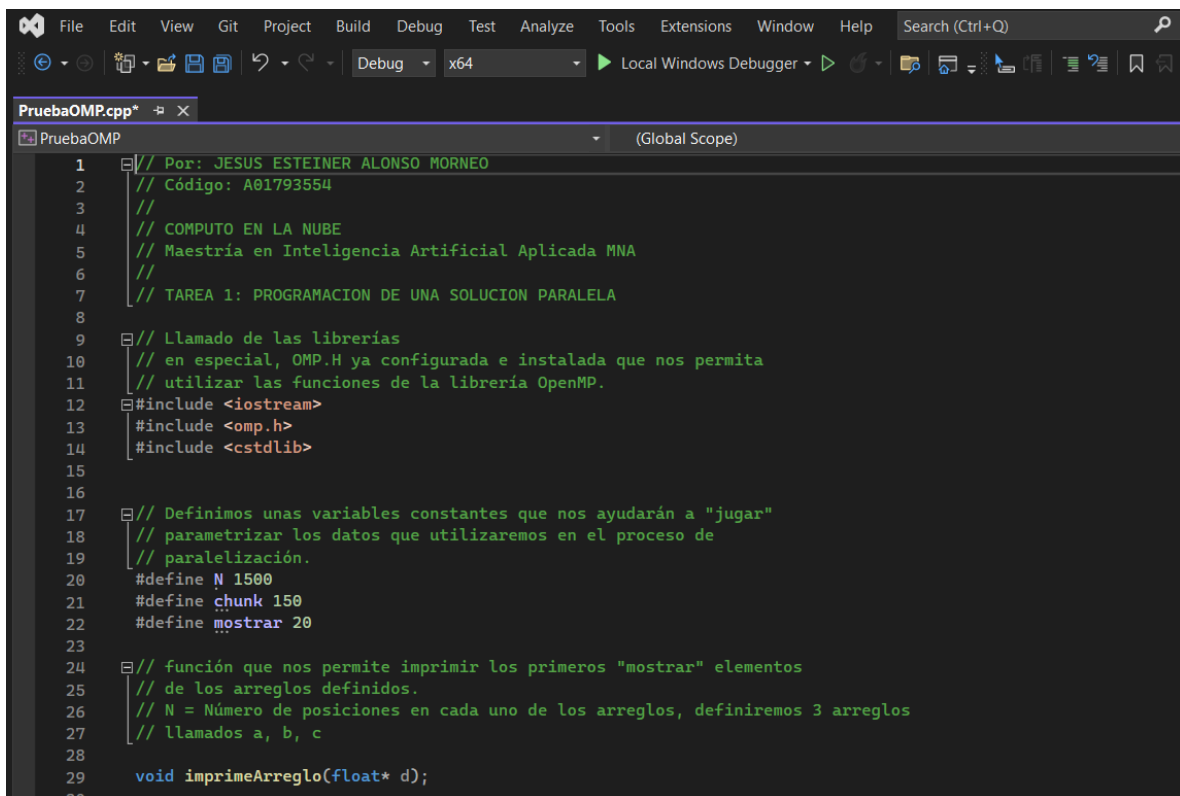
Para mucha más información acerca de OpenMP favor referirse a www.openmp.org.

Liga del repositorio del proyecto en Github:

<https://github.com/PosgradoMNA/actividades-de-aprendizaje-JesusAlonsoTecM/blob/main/PruebaOMP.cpp>

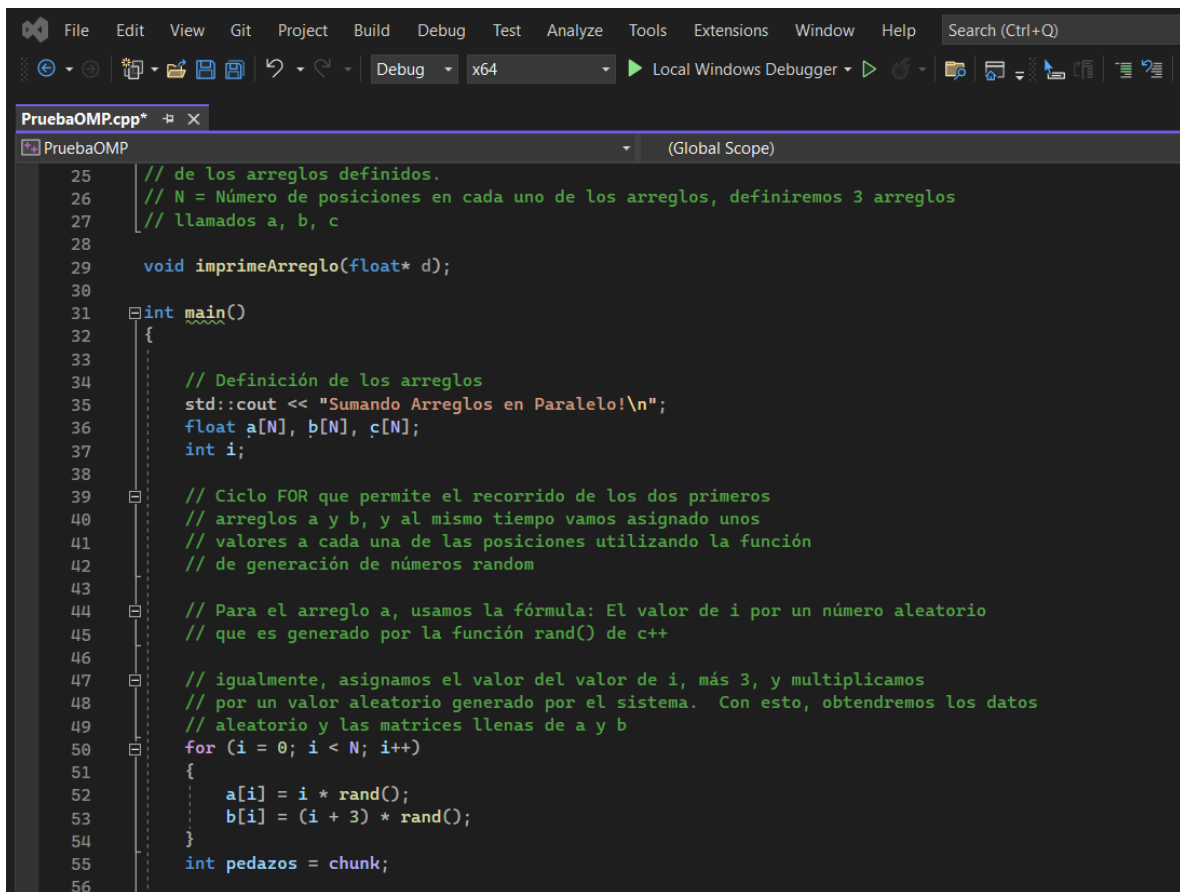
Explicación del código a través de capturas de pantalla

Pantalla 1: Llamado de librerías y definición de constantes



```
1 // Por: JESUS ESTEINER ALONSO MORNEO
2 // Código: A01793554
3 //
4 // COMPUTO EN LA NUBE
5 // Maestría en Inteligencia Artificial Aplicada MNA
6 //
7 // TAREA 1: PROGRAMACION DE UNA SOLUCION PARALELA
8
9 // Llamado de las librerías
10 // en especial, OMP.H ya configurada e instalada que nos permita
11 // utilizar las funciones de la librería OpenMP.
12 #include <iostream>
13 #include <omp.h>
14 #include <cstdlib>
15
16
17 // Definimos unas variables constantes que nos ayudarán a "jugar"
18 // parametrizar los datos que utilizaremos en el proceso de
19 // paralelización.
20 #define N 1500
21 #define chunk 150
22 #define mostrar 20
23
24 // función que nos permite imprimir los primeros "mostrar" elementos
25 // de los arreglos definidos.
26 // N = Número de posiciones en cada uno de los arreglos, definiremos 3 arreglos
27 // llamados a, b, c
28
29 void imprimeArreglo(float* d);
30
```

Pantalla 2: Definición y creación de 3 arreglos float, con N número de posiciones, en nuestro caso inicial, colocaremos 1500. Adicionalmente, se utiliza un FOR para recorrer los arreglos a y b, y llenarlos con valores aleatorios usando la función rand() de c++.



```
25 // de los arreglos definidos.
26 // N = Número de posiciones en cada uno de los arreglos, definiremos 3 arreglos
27 // llamados a, b, c
28
29 void imprimeArreglo(float* d);
30
31 int main()
32 {
33
34     // Definición de los arreglos
35     std::cout << "Sumando Arreglos en Paralelo!\n";
36     float a[N], b[N], c[N];
37     int i;
38
39     // Ciclo FOR que permite el recorrido de los dos primeros
40     // arreglos a y b, y al mismo tiempo vamos asignando unos
41     // valores a cada una de las posiciones utilizando la función
42     // de generación de números random
43
44     // Para el arreglo a, usamos la fórmula: El valor de i por un número aleatorio
45     // que es generado por la función rand() de c++
46
47     // igualmente, asignamos el valor del valor de i, más 3, y multiplicamos
48     // por un valor aleatorio generado por el sistema. Con esto, obtendremos los datos
49     // aleatorio y las matrices llenas de a y b
50     for (i = 0; i < N; i++)
51     {
52         a[i] = i * rand();
53         b[i] = (i + 3) * rand();
54     }
55     int pedazos = chunk;
56 }
```

Pantalla 3: Explicación de la instrucción más importante dentro del código #pragma

```
PruebaOMP.cpp* X
(Global Scope)
61 int pedazos = chunk;
62
63 // El estándar OpenMP, API que expresa paralelismo multihilo en sistemas de memoria compartida
64 // Componentes:
65
66 // - Directivas #pragma de compilación
67 // - informan al compilador para optimizar código siguiente
68 //
69 // #pragma omp <directiva> {<cláusula>}* <\n>
70 //
71 // funciones de la librería
72 // variables de entorno
73 //
74 // De esta manera, un bucle es fácilmente paralelizable en Openmp
75 //
76 // Reglas:
77 // - No dependencia entre iteraciones
78 // - Prohibidas instrucciones break, exit(), goto...
79 // - Forma CANÓNICA -> for (i=INICIO; i {>,<,>=,<=}, i++,i--,++i,--i
80 // - Una sola instrucción en un bloque
81 //
82 // Cláusulas utilizadas:
83 // - shared(a, b, c, pedazos) -> Especifica que las variables, a, b, c y pedazos
84 // deben compartirse entre todos los subprocesos.
85 // - private(i) -> Existe una copia de las variables i en el for para cada hilo,
86 // es decir, Especifica que cada subproceso debe tener su propia
87 // instancia de una variable.
88 // - schedule(static, pedazos) -> Cómo se distribuyen los hilos en cada trabajo,
89 // --> Se aplica a la directiva for.
90 //
91 // --> schedule(<tipo>[,<tamaño>])
92 // --> Para este caso, el uso del tipo <static> muestra las iteraciones
93 // --> contiguas por cada hilo, es decir, el valor chunk, 150 en nuestro caso.
94 #pragma omp parallel for shared(a, b, c, pedazos) private(i) schedule(static, pedazos)
```

Pantalla 4: Asignación del resultado de la suma de los arreglos a y b al arreglo c, y adicionalmente, se imprima el resultado de la sumatoria mostrando los primeros “mostrar” lugares, en nuestro caso 20 por el momento.

```
95
96 // Recorrido de los arreglos, asignado la sumatoria de los arreglos a y b al arreglo c
97 // en la misma posición
98 for (i = 0; i < N; i++)
99     c[i] = a[i] + b[i];
100
101 // Impresión de resultados al llamar a la función imprimeArreglo pasándole como parámetro
102 // el arreglo a imprimir, adicionalmente, usará el valor de la constante "mostrar" definida en
103 // valor de 20, para que se pueda observar por pantalla los resultados de la suma
104 std::cout << "Imprimiendo los primeros " << mostrar << " valores del arreglo a: " << std::endl;
105 imprimeArreglo(a);
106
107 std::cout << "Imprimiendo los primeros " << mostrar << " valores del arreglo b: " << std::endl;
108 imprimeArreglo(b);
109
110 std::cout << "Imprimiendo los primeros " << mostrar << " valores del arreglo c: " << std::endl;
111 imprimeArreglo(c);
112 }
```

Pantalla 5: Definición de la función imprimeArreglo()

```
114 void imprimeArreglo(float* d)
115 {
116     for (int x = 0; x < mostrar; x++)
117         std::cout << d[x] << " - ";
118     std::cout << std::endl;
119 }
```

CAPTURAS DE PANTALLA CON LOS RESULTADOS de ejecución del algoritmo:

Ejecución 1, con parámetros:

```
// Definimos unas variables constantes que nos ayudarán a "jugar"
// parametrizar los datos que utilizaremos en el proceso de
// paralelización.
#define N 1500
#define chunk 150
#define mostrar 20
```

Resultados:

```
en especial, OMP.H ya configurada e instalada que nos permita
utilizar las funciones de la libreria OpenMP.
#include <iostream>
#include <omp.h>
#include <cstdlib>

// Definición de constantes
const int N = 1500; // Número de elementos del arreglo
const int chunk = 100; // Tamaño del chunk para la división del trabajo

// Función para imprimir los primeros 20 valores de un arreglo
void imprimirArreglo(const int a[], int n) {
    for (int i = 0; i < 20; i++) {
        cout << a[i] << " ";
        if (i % 10 == 9) cout << "\n";
    }
}

// Función para generar un arreglo de números aleatorios
void generarArreglo(int a[]) {
    for (int i = 0; i < N; i++) {
        a[i] = rand() % 1000000;
    }
}

// Función para calcular la suma de los elementos de un arreglo en paralelo
double calcularSumaParalela(const int a[], int n) {
    double suma = 0.0;
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        suma += a[i];
    }
    return suma;
}

// Función principal
int main() {
    // Generar el arreglo
    int a[N];
    generarArreglo(a);

    // Imprimir los primeros 20 valores del arreglo
    imprimirArreglo(a, N);

    // Calcular la suma del arreglo en paralelo
    double suma = calcularSumaParalela(a, N);

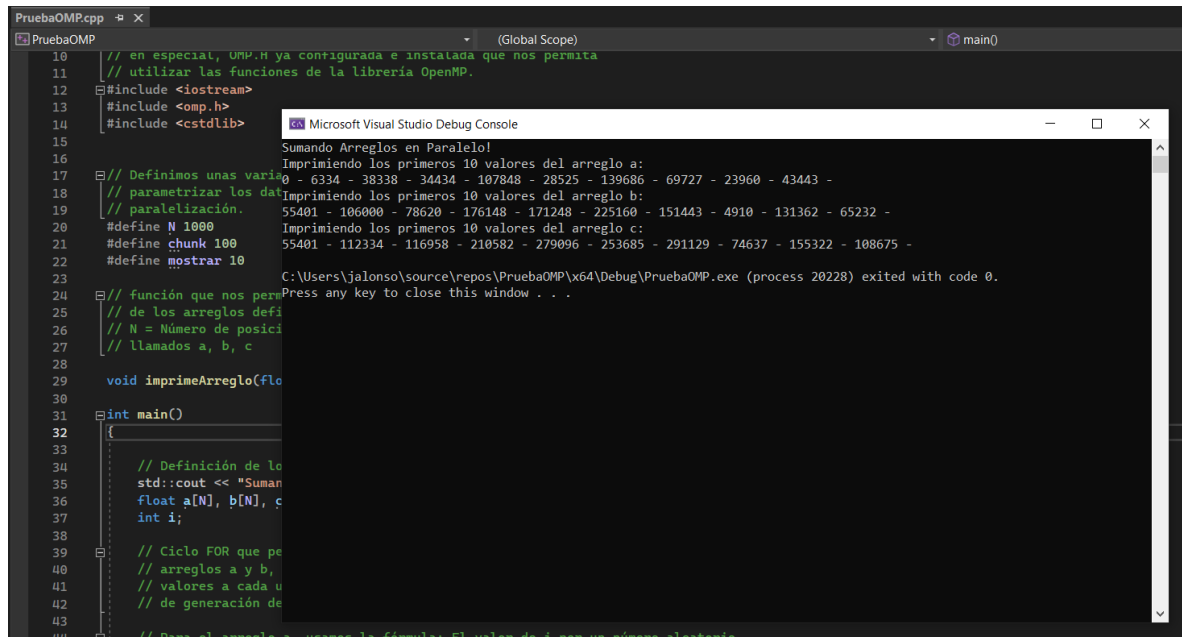
    // Imprimir el resultado
    cout << "Suma total: " << suma << endl;

    return 0;
}
```

Ejecución 2, con parámetros:

```
17 // Definimos unas variables constantes que nos ayudarán a "jugar"
18 // parametrizar los datos que utilizaremos en el proceso de
19 // paralelización.
20 #define N 1000
21 #define chunk 100
22 #define mostrar 10
23
```

Resultados:



The screenshot shows the Visual Studio IDE with the file 'PruebaOMP.cpp' open. The code defines constants for N (1000), chunk (100), and mostrar (10). It includes OpenMP headers and defines a function 'imprimeArreglo' to print array elements. The main function uses OpenMP to parallelize the printing of three arrays (a, b, and c) in chunks of 100 elements, displaying the first 10 elements of each. The Debug Console shows the output of the program, which prints the first 10 elements of each array in parallel.

```
Sumando Arreglos en Paralelo!
Imprimiendo los primeros 10 valores del arreglo a:
0 - 6334 - 38338 - 34434 - 107848 - 28525 - 139686 - 69727 - 23960 - 43443 -
Imprimiendo los primeros 10 valores del arreglo b:
55401 - 106000 - 78620 - 176148 - 171248 - 225160 - 151443 - 4910 - 131362 - 65232 -
Imprimiendo los primeros 10 valores del arreglo c:
55401 - 112334 - 116958 - 210582 - 279096 - 253685 - 291129 - 74637 - 155322 - 108675 -
C:\Users\jalonso\source\repos\PruebaOMP\x64\Debug\PruebaOMP.exe (process 20228) exited with code 0.
Press any key to close this window . . .
```

REFLEXION SOBRE LA PROGRAMACION PARALELA

Ya hemos logrado comprobar que la programación paralela tiene muchas ventajas, por supuesto también un par de desventajas, pero es un recurso que podemos aprovechar en gran variedad de procesos permitiéndonos ahorrar tiempo y recursos cuando este es posible. Sabemos ahora, que la utilización por ejemplo de la interfaz OpenMP como un modelo para la programación paralela para memoria compartida y memoria distribuida en multiprocesadores, nos permite tener a disposición varios hilos de ejecución concurrentes que dan acceso a variables alojadas en zonas de memoria compartida, y esto facilita enormemente la obtención de resultados mucho más rápido, y de una forma flexible acelerando la ejecución de un programa como resultado de la paralelización. La programación en paralelo ya es hoy día un paradigma de programación dominante, por supuesto principalmente en arquitecturas de computadores que mantienen procesadores multinúcleo. Entendemos sin embargo, que hay una curva de aprendizaje mucho más alta en la programación paralela que lleva a incrementar su complejidad de

implementación, así como incrementos en energía y algunos otros aspectos, pero ello no detendrá que cada día más, este paradigma sea y se convierta aún más, en el paradigma dominante en la programación de computadoras.