Programación y Análisis de Algoritmos | Tarea Intro OpenMP

Aguirre Calzadilla César Miguel

31 de octubre de 2024

Códigos

Todo el código escrito para esta tarea será anexado en un archivo .zip. Dentro se encuentran las rutinas escritas para la tarea así como comentarios sobre las mismas.

El objetivo de la tarea es implementar un código secuencial, y otro paralelo, para resolver los siguientes problemas.

Problema 1

La suma de los elementos de un vector de tamaño N

El problema en este caso se enfoca en resolver la suma, elemento a elemento, de múltiples vectores de igual tamaño, a saber, N. La solución propuesta itera sobre cada entrada de los vectores colocados como ejemplo (los cuales se pueden modificar), sumando los elementos correspondientes, para así almacenar el resultado en un nuevo vector.

En el caso secuencial, se realiza la suma de forma lineal. Es decir, se itera sobre cada posición de los vectores, uno después del otro. Naturalmente, no se está implementado procesamiento de múltiples núcleos.

Podríamos decir, que la forma secuencial es quizás más sencilla de interpretar y seguir para la mayoría de los programadores. En general, los ejemplos vistos aquí de manera secuencial servirán para datos compactos, y de fácil manipulación.

Por otra parte, en el caso paralelo, se puede aprovechar la división del problema en los distintos núcleos para así aprovechar la potencial de los sistemas multi-core (algo habitual para la bastan mayoría de las computadoras actuales). Esto provoca que múltiples núcleos del procesador puedan acelerar el tiempo de ejecución de un problema. En este caso en particular, no estamos implementando vectores muy grandes, pero se podría hacer.

La principal diferencia entre ambos códigos está en la implementación de la suma. En el caso secuencial, esto se ve así:

```
1  // Sumar elementos de cada vector secuencialmente
2
3  for (size_t i = 0; i < n; i++) {
4      for (const auto& vec : vectores) {
5          resultado[i] += vec[i];
6      }
7  }
}8
```

Listing 1: Código secuencial en C++ para la suma de vectores

Mientras que en el caso paralelo se ve así:

```
1
2  // Paralelizar la suma con OpenMP
3  #pragma omp parallel for
4  for (size_t i = 0; i < n; i++) {
5      for (const auto& vec : vectores) {
6          // Seccion critica para la impresion sincronizada
7          #pragma omp critical
8          {
9              int thread_id = omp_get_thread_num();
10             std::cout << "El hilo " << thread_id << " esta sumando
   el elemento vec[" << i << "] = " << vec[i] << std::endl;
11         }
12         resultado[i] += vec[i];
13     }
14 }
```

Listing 2: Código paralelo en C++ para la suma de vectores

Podemos ver que la lógica es la misma, lo único que se está haciendo es llamar a OpenMP, en el caso paralelo, para paralelizar la suma de los vectores. En realidad, lo que se está paralelizando son los cilcos for que abordan el problema.

El primer ciclo for (`for (size_t i = 0; i < n; i++) { ... }`) itera sobre cada posición de los vectores. Aquí, `i` es un índice que inicializa en 0 y recorre por completo al vector. Por otra parte, el segundo ciclo for (`for (const auto& vec : vectores) { ... }`) realiza la iteración sobre cada vector.

Problema 2

Multiplicación de dos matrices cuadradas de tamaño NxN

En este caso, se nos solicita atacar el problema de la multiplicación de matrices.

En el caso del código secuencial, naturalmente no se realiza ningún tipo de paralelización. La lógica en ambos problemas es la misma: se realiza multiplicación por bloques.

En sí, la idea es sencilla. Se busca dividir la matriz en bloques más pequeños que realicen multiplicación entre los mismos bloques. Esta técnica es ampliamente implementada para mejorar el rendimiento del algoritmo. Lo primero que se hace es dividir las matrices en submatrices $BS \times Bs$. Para cada combinación de bloques de A y B, se calcula el producto, y el resultado se almacena en C. Al final, los resultados de la multiplicación de bloques se suma, para obtener la multiplicación total de las matrices A y B.

La idea del código puede encontrarse en documentación oficial de OpenMP: OpenMP Application Programming Interface.

```
1
2 Funcion para multiplicar dos matrices usando OpenMP
3  void matmul_depend(int N, int BS, float** A, float** B, float** C)
   {
4      int i, j, k, ii, jj, kk;
5
6      // Inicializar la matriz resultado C en ceros
7      #pragma omp parallel for collapse(2) // Paralelizar la
   inicializaci n
8      for (i = 0; i < N; i++) {
9          for (j = 0; j < N; j++) {
10             C[i][j] = 0.0f;
11         }
12     }
13
14     // Multiplicacion de matrices en bloques
15     for (i = 0; i < N; i += BS) {
16         for (j = 0; j < N; j += BS) {
17             for (k = 0; k < N; k += BS) {
18                 // Crear una tarea para cada bloque
19                 #pragma omp task private(ii, jj, kk)
20                 {
21                     for (ii = i; ii < i + BS; ii++) {
22                         for (jj = j; jj < j + BS; jj++) {
23                             for (kk = k; kk < k + BS; kk++) {
24                                 C[ii][jj] += A[ii][kk] * B[kk][jj];
25                             }
26                         }
27                     }
28                 }
29             }
30         }
31     }
32 }
```

Listing 3: Código paralelo en C++ para multiplicación de matrices por bloques

```
1
2   // Funcion para multiplicar dos matrices de forma secuencial
3  void matmul_depend(int N, int BS, float** A, float** B, float** C) {
4   int i, j, k, ii, jj, kk;
5
6   // Inicializar la matriz resultado C en ceros
7   for (i = 0; i < N; i++) {
8       for (j = 0; j < N; j++) {
9           C[i][j] = 0.0f;
10      }
11  }
12
13  // Multiplicacion de matrices en bloques (sin paralelizaci n)
14  for (i = 0; i < N; i += BS) {
15      for (j = 0; j < N; j += BS) {
16          for (k = 0; k < N; k += BS) {
17              for (ii = i; ii < i + BS; ii++) {
18                  for (jj = j; jj < j + BS; jj++) {
19                      for (kk = k; kk < k + BS; kk++) {
20                          C[ii][jj] += A[ii][kk] * B[kk][jj];
21                      }
22                  }
23              }
24          }
25      }
26  }
27 }
```

Listing 4: Código secuencial en C++ para multiplicación de matrices por bloques

Creo que vale la pena mencionar que en este código se utiliza la línea `#pragma omp parallel for collapse (2)`, la cual indica al compilador que combine los dos bucles anidados en un solo bucle durante la paralelización. Esto significa que, en lugar de tener un bucle externo y uno interno, lo que pasa es que se crea un solo bucle que abarca las iteraciones de ambos. En teoría, esto permite una mejor distribución en el cómputo de la multiplicación.

También nos encontramos con `#pragma omp task private (ii, jj, kk)`. Esta declaración vuelve privadas a las variables (ii, jj, kk). Es decir, cada hilo que ejecute la tera tendrá su propia copia de (ii, jj, kk), y la modificación de las mismas dentro de cada hilo no afectará a las copias dentro de otros hilos de paralelización.

Utilizar variables privadas evita que múltiples hilos intenten acceder, o incluso que modifiquen, a las mismas variables al mismo tiempo.

Problema 3a

Dado un vector de números reales V de tamaño N, Resuelve lo siguiente: sea $S_v[i] = V[i] + V[i+1] para i = 0, 1, 2, .., N-2$ con S_v otro vector de tamaño $N-1$.

En este caso, se solicita calcular las sumas de los pares de elementos adyacentes dentro de un vector V, y almacenar los resultados en un vector S_v.

Pensemos en el siguiente ejemplo:

Sea el vector $\vec{v} = (1, 2, 3, 4, 5)$, entonces la suma de sus elementos adyacentes serían $S_0 = v_0 + v_1 = 1 + 2 = 3$, $S_1 = v_1 + v_2 = 2 + 3 = 5$, $S_2 = v_2 + v_3 = 3 + 4 = 7$ y $S_3 = v_3 + v_4 = 4 + 5 = 9$. Lo que daría como resultado a $S_v = (3, 5, 7, 9)$.

```
    // Paralelizar el bucle utilizando OpenMP
#pragma omp parallel for
for (int i = 0; i < N - 1; i++) {
    int thread_id = omp_get_thread_num(); // Obtener el ID del thread
    Sv[i] = V[i] + V[i + 1]; // C lculo

    // Imprimir que hace cada thread en una seccion critica
    #pragma omp critical
    {
        std::cout << "Thread " << thread_id << " esta calculando Sv[" << i << "] = "
                  << V[i] << " + " << V[i + 1] << " = " << Sv[i] << std::endl;
    }
}
```

Listing 5: Código paralelo en C++ para suma de elementos adyacentes dentro de un vector.

```
    // Bucle secuencial para calcular Sv
for (int i = 0; i < N - 1; i++) {
    Sv[i] = V[i] + V[i + 1]; // C lculo

    // Imprimir qu  se est  calculando
    std::cout << "Calculando Sv[" << i << "] = "
              << V[i] << " + " << V[i + 1] << " = " << Sv[i] << std::endl;
}
```

Listing 6: Código secuencial en C++ para suma de elementos adyacentes dentro de un vector.

Para el caso paralelo, se utiliza `#pragma omp parallel for` para que el bucle for se paralelice. Entonces, múltiples hilos ejecutarán las iteraciones para construir el vector S_v. Dentro del bucle, cada hilo suma dos elementos adyacentes del vector V, para luego almacenarlo en el S_v.

Además, `#pragma omp critical` se utiliza para que solo un hilo a la vez pueda ejecutar el código dentro de la sección "crítica". En este caso, se utiliza para que la salida del código se imprima de manera ordenada.

Problema 3b

Dado un vector de números reales V de tamaño N, Resuelve lo siguiente: sea $S_w[i] = \frac{V[i+1]+V[i-1]}{2} parai = 0, 1, 2, .., N-2$ con S_w otro vector de tamaño $N-2$.

Contrario al caso anterior, en este problema se calcular el promedio de los elementos $V[i+1] + V[i-1]$ dado un vector V.

```
// Calculo de S2 con OpenMP
#pragma omp parallel for
for (int i = 1; i < N - 1; i++) {
    // Cada thread calcula S2[i - 1] como el promedio de los elementos V[i - 1] y V[i + 1]
    S2[i - 1] = (V[i - 1] + V[i + 1]) / 2.0;

    // Imprimir el calculo realizado por cada thread
    #pragma omp critical // Asegura que solo un thread imprima a la vez
    std::cout << "Thread " << omp_get_thread_num()
              << " esta calculando S2[" << (i - 1) << "] = ("
              << V[i - 1] << " + " << V[i + 1] << ") / 2 = "
              << S2[i - 1] << std::endl;
}
```

Listing 7: Código paralelo en C++ para calcular el promedio de los elementos dentro de un vector.

```
for (int i = 1; i < N - 1; i++) {
    // Calcular S2[i - 1] como el promedio de los elementos V[i - 1] y V[i + 1]
    S2[i - 1] = (V[i - 1] + V[i + 1]) / 2.0;

    // Imprimir el calculo realizado
    std::cout << "Calculando S2[" << (i - 1) << "] = ("
              << V[i - 1] << " + " << V[i + 1] << ") / 2 = "
              << S2[i - 1] << std::endl;
}
```

Listing 8: Código secuencial en C++ para suma de elementos adyacentes dentro de un vector.

Problema 4a

Dadas dos matrices A y B de tamaño $N \times M$, con valores enteros positivos, programar: C1(i,j) = A(i,j) + B(N-i-1, M-j-1) para $i = 0, .., N-1$ y $j = 0, .., M-1$, con C1 otra matriz de tamaño $N \times M$

Lo que se nos está pidiendo para este problema es realizar una suma de matrices, a saber, la A y B, que almacenan el resultado en una nueva matriz C_1. Para esta nueva matriz, cada elemento $C_1[i][j]$ es calculado como la suma de $A[i][j] + B[N-i-1][M-j-1]$.

Esto realiza una inversión en los índices, para el caso de matrices $\mathbf{M}_{2x2}$ se vería como:

$$\begin{cases} C1[0][0] = A[0][0] + B[1][1] \\ C1[0][1] = A[0][1] + B[1][0] \\ C1[1][0] = A[1][0] + B[0][1] \\ C1[1][1] = A[1][1] + B[0][0] \end{cases}$$

En este caso, el enfoque paralelo, podemos aprovechar computar múltiples elementos al mismo tiempo. En casos de ejemplos con matrices muy, muy grandes, esto pude reducir el tiempo de cómputo significativamente.

```
1  // Calculo de C1 utilizando OpenMP
2  #pragma omp parallel for collapse(2)
3  for (int i = 0; i < N; i++) {
4      for (int j = 0; j < M; j++) {
5          C1[i][j] = A[i][j] + B[N - i - 1][M - j - 1];
6
7          // Imprimir el calculo realizado por cada thread
8          #pragma omp critical // Asegura que solo un thread imprima
   a la vez
9          std::cout << "Thread " << omp_get_thread_num()
10                   << " esta calculando C1[" << i << "][" << j << "]
    = "
11                   << A[i][j] << " + " << B[N - i - 1][M - j - 1]
12                   << " = " << C1[i][j] << std::endl;
13     }
14 }
```

Listing 9: Código paralelo en C++.

```
1  // Calculo de C1 de forma secuencial
2  for (int i = 0; i < N; i++) {
3      for (int j = 0; j < M; j++) {
4          C1[i][j] = A[i][j] + B[N - i - 1][M - j - 1];
5
6          // Imprimir el calculo realizado en cada paso
7          std::cout << "Calculando C1[" << i << "][" << j << "] = "
8                    << A[i][j] << " + " << B[N - i - 1][M - j - 1]
9                    << " = " << C1[i][j] << std::endl;
10     }
11 }
```

Listing 10: Código secuencial en C++.

Problema 4b

Dadas dos matrices A y B de tamaño $N \times M$, con valores enteros positivos, programar: $C2(i,j) = \alpha A(i,j) + (1+\alpha)B(i,j)$ para $i = 0,..,N-1$ y $j = 0,..,M-1$, con α un real entre [0, 1] y C2 otra matriz de tamaño $N \times M$

En este caso, se busca construir una combinación lineal de las matrices A y B, a través de un escalar en R con valor en [0,1]. Nuevamente, para casos de matrices muy grandes, la paralelización suena a una buena idea.

```cpp
// Calculo de C2 utilizando OpenMP
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        C2[i][j] = alpha * A[i][j] + (1 - alpha) * B[i][j];

        // Imprimir el calculo realizado por cada thread
        #pragma omp critical // Asegura que solo un thread imprima a la vez
        std::cout << "Thread " << omp_get_thread_num()
                  << " est  calculando C2[" << i << "][" << j << "] = "
                  << alpha << " * " << A[i][j] << " + "
                  << "(1 - " << alpha << ") * " << B[i][j] << " = "
                  << C2[i][j] << std::endl;
    }
}
```

Listing 11: Código paralelo en C++.

```cpp
// Calculo de C2 de forma secuencial
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        C2[i][j] = alpha * A[i][j] + (1 - alpha) * B[i][j];

        // Imprimir el calculo realizado en cada paso
        std::cout << "Calculando C2[" << i << "][" << j << "] = "
                  << alpha << " * " << A[i][j] << " + "
                  << "(1 - " << alpha << ") * " << B[i][j] << " = "
                  << C2[i][j] << std::endl;
    }
}
```

Listing 12: Código secuencial en C++.

Conclusiones

Con estos casos podemos llegar a diferentes conclusiones. Para empezar, que las operaciones con vectores y matrices pueden optimizarse, en el sentido del tiempo de computo, sobre todo cuando trabajamos con cantidades grandes de datos. A pesar de que en estos casos trabajamos con muestras pequeñas, se puede escalar fácilmente a casos mucho más grandes.

OpenMP permite dividir el procesamiento de los datos, aprovechando la estructura de múltiples núcleos en las computadoras. Esto es algo realmente común en las computadoras actuales. Hasta los celulares ya vienen multinucleo. La flexibilidad de OpenMP puede observarse en el uso de paralelización en la suma de vectores y multiplicación de matrices por bloques. Como un ejemplo, usando diferentes declaraciones de #pragma en la multiplicación de matrices, por ejemplo, permite a cada núcleo trabajar en un bloque independiente, evitando así conflictos entre hilos (como con #pragma omp task).

Aunque la implementación secuencial es con la que la mayoría de los programadores están acostumbrados a trabajar, sobre todo cuando son primerizos, mejorar el rendimiento del computo en diversas industrias y cómputo científico puede ser una gran diferencia entre entregar un proyecto en unos días o en unas horas.

Finalmente, me gustaría listar algunas declaraciones de OpenMP que utilicé en esta tarea.

Declaraciones `#pragma`

- `#pragma omp parallel for collapse(2)`
  Utilizada para paralelizar bucles anidados con OpenMP. La opción `collapse(2)` combina los dos bucles más externos, dividiendo las iteraciones entre hilos para mejorar la eficiencia.

- `#pragma omp critical`
  Protege una sección de código para que solo un hilo la ejecute a la vez, evitando conflictos en operaciones críticas como la impresión de resultados en consola.

- `#pragma omp task private(...)`
  Declara una tarea independiente para su ejecución asíncrona en OpenMP, con variables privadas para cada hilo. En el contexto del código, se utiliza para dividir el cálculo de bloques en la multiplicación de matrices.

- `#pragma omp parallel for`
  Utilizada para paralelizar bucles con OpenMP, distribuyendo las iteraciones entre los hilos disponibles, como en la suma de vectores o el cálculo de promedios.