Universidad Católica San Pablo

Ejercicios del Chapter 4: C++11 MULTITHREADING

Computer Science — Parallel and Distributed Computing

Harold Alejandro Villanueva Borda

1. Ejercicio 2

Considere dos matrices cuadradas $A, B \in \mathbb{R}^{n \times n}$ cada una de ancho $n = 1024$. A continuación queremos calcular su suma $C = A + B$ mediante la suma indexada de entradas $C_{ij} = A_{ij} + B_{ij}$ para todo $i, j \in \{0, \ldots, n-1\}$. Implemente un programa paralelo que utilice p subprocesos para calcular las entradas de C. Experimente con diferentes patrones de distribución de subprocesos estáticos.

1. **Configuración Inicial**:
   - Definimos el tamaño de las matrices $n = 1024$.
   - Inicializamos las matrices A y B con valores aleatorios o predefinidos.
   - Definimos la matriz C que almacenará el resultado.
2. **Distribución de Trabajo**:
   - Dividimos el trabajo entre p hilos. Cada hilo procesará un bloque de filas de las matrices A y B y calculará las filas correspondientes de C.
3. **Creación y Ejecución de Hilos**:
   - Creamos y lanzamos los hilos para realizar la suma de las matrices en paralelo.
4. **Unión de Hilos**:
   - Esperamos a que todos los hilos completen su trabajo y se unan al hilo principal.
5. **Código en C++11**:

```cpp
#include <iostream>
#include <vector>
#include <thread>
#include <random>  // Para std::random_device y std::mt19937

const int n = 1024;  // Tama o de las matrices
const int p = 8;     // N mero de hilos

// Funci n que cada hilo ejecutar  para sumar las matrices
void sum_matrices(int thread_id, const std::vector<std::vector<int>>& A, const std::vector<
    std::vector<int>>& B, std::vector<std::vector<int>>& C) {
    int rows_per_thread = n / p;
    int start_row = thread_id * rows_per_thread;
```

```cpp
    int end_row = (thread_id + 1) * rows_per_thread;

    for (int i = start_row; i < end_row; ++i) {
        for (int j = 0; j < n; ++j) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}

int main() {
    // Inicializaci n de matrices A y B con valores aleatorios
    std::vector<std::vector<int>> A(n, std::vector<int>(n));
    std::vector<std::vector<int>> B(n, std::vector<int>(n));
    std::vector<std::vector<int>> C(n, std::vector<int>(n));

    std::random_device rd;  // Generador de n meros aleatorios
    std::mt19937 gen(rd()); // Motor de generaci n de n meros aleatorios
    std::uniform_int_distribution<> dis(0, 99); // Distribuci n uniform

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            A[i][j] = dis(gen);
            B[i][j] = dis(gen);
        }
    }

    // Vector para almacenar los hilos
    std::vector<std::thread> threads;

    // Creaci n y lanzamiento de hilos
    for (int i = 0; i < p; ++i) {
        threads.emplace_back(sum_matrices, i, std::ref(A), std::ref(B), std::ref(C));
    }

    // Uni n de hilos
    for (int i = 0; i < p; ++i) {
        threads[i].join();
    }

    // mostramos una parte de la matriz C
    std::cout << "C[0][0] = " << C[0][0] << std::endl;
    std::cout << "C[n-1][n-1] = " << C[n-1][n-1] << std::endl;

    return 0;
}
```

Implementation

Explicación del Código

- **Inicialización de Matrices**: Las matrices A y B se inicializan con valores aleatorios usando `std::rand()`. La matriz C se inicializa para almacenar los resultados de la suma.
- **Función de Suma**: La función `sum_matrices` toma un `thread_id` para identificar qué parte de las filas debe procesar. Cada hilo calcula su porción correspondiente de la suma.

- **Lanzamiento y Unión de Hilos**: Se crean p hilos y se les asigna la tarea de sumar las matrices. Luego, el hilo principal espera a que todos los hilos terminen su trabajo con `join()`.

Consideraciones sobre la Distribución de Hilos

- **Distribución Estática**: En esta implementación, cada hilo procesa un bloque de filas de tamaño fijo. Esto se conoce como distribución estática y es fácil de implementar y eficiente si las filas tienen un tiempo de procesamiento similar.
- **Balanceo de Carga**: Si algunas filas fueran más costosas de procesar que otras, una distribución dinámica (donde los hilos solicitan nuevas filas para procesar después de terminar las actuales) podría ser más eficiente. Esto evitaría que algunos hilos terminen mucho antes que otros.

2. Ejercicio 4

Considera el siguiente fragmento de código:

```
#include <iostream>
#include <thread>
int main () {
    auto child = [] () -> void {
        std::cout << "child" << std::endl;
    };
    std::thread thread(child);
    thread.detach();

    std::cout << "parent" << std::endl;
}
```

En la mayoría de los casos, la salida del programa es exclusivamente “parent” ya que llegamos al final de la función principal antes de la declaración de impresión en el cierre del hijo. Sin embargo, el uso de `thread.join();` en lugar de `thread.detach();` resulta en la salida “child” seguida de “parent”. Ahora supón que tuviéramos que implementar nuestro propio método join usando variables de condición. Procederíamos de la siguiente manera:

a. Introducimos una variable booleana global `done = false`. Además, necesitamos un mutex `m` y una variable de condición `c`.

b. Después de la declaración de impresión en child, establecemos `done = true` en un alcance bloqueado y luego notificamos la variable de condición `c`.

c. Nuestro método join personalizado realiza una espera condicional en un alcance bloqueado siempre que `done == false`.

Implementa el método join personalizado descrito. Prueba diferentes implementaciones: experimenta con el bloqueo explícito de mutexes y bloqueos de alcance.

1. Explicación del comportamiento:
    - En la mayoría de los casos, la salida del programa es exclusivamente "parentza que se llega al final de la función `main` antes de que se ejecute la declaración de impresión en el cierre `child`.

- El uso de `thread.join();` en lugar de `thread.detach();` resulta en la salida çhild"seguida de "parent".

2. Implementación del Método `join` Personalizado:

a. Introducción de variables globales:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

bool done = false;
std::mutex m;
std::condition_variable c;
```

- `done`: Variable booleana global que indica si el hilo hijo ha terminado su ejecución.
- `m`: Mutex global para sincronizar el acceso a `done`.
- `c`: Variable de condición global para notificar cuando el hilo hijo ha terminado.

b. Modificación de la función del hilo hijo:

```
auto child = [] () -> void {
   std::cout << "child" << std::endl;
   std::unique_lock<std::mutex> lock(m);
   done = true;
   c.notify_one();
};
```

- Después de imprimir çhild", se adquiere un bloqueo único (`std::unique_lock`) en el mutex `m`.
- Se establece `done` a `true` dentro del alcance del bloqueo.
- Se notifica a la variable de condición `c` que el hilo ha terminado.

c. Implementación del método `join` personalizado:

```
void custom_join() {
   std::unique_lock<std::mutex> lock(m);
   c.wait(lock, []{return done;});
}
```

- `custom_join` adquiere un bloqueo único en el mutex `m`.
- La función `c.wait` espera a que `done` sea `true`, liberando el bloqueo temporalmente y volviéndolo a adquirir cuando se despierta.

d. Modificación de la función `main`:

```
int main() {
   auto child = [] () -> void {
       std::cout << "child" << std::endl;
       std::unique_lock<std::mutex> lock(m);
       done = true;
       c.notify_one();
   };
   std::thread thread(child);
   custom_join();
   std::cout << "parent" << std::endl;
}
```

- Se inicia el hilo `child` y luego se llama a `custom_join` para esperar a que el hilo `child` termine antes de continuar.
- Después de que `custom_join` retorna, se imprime "parent".

3. Prueba de Diferentes Implementaciones:

   Con bloqueo explícito:

```
void custom_join() {
   std::unique_lock<std::mutex> lock(m);
   while (!done) {
       c.wait(lock);
   }
}
```

- Aquí, `custom_join` usa un ciclo `while` para esperar a que `done` sea `true`.

4. Con bloqueos de alcance:

```
void custom_join() {
   std::lock_guard<std::mutex> lock(m);
   c.wait(m, []{return done;});
}
```

- Esta versión no es válida ya que `std::condition_variable::wait` requiere un `std::unique_lock`.

5. Código Completo con Implementación del `join` Personalizado:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

bool done = false;
std::mutex m;
std::condition_variable c;
```

```cpp

//bloqueo join personalizado
void custom_join1() {
    std::unique_lock<std::mutex> lock(m);
    c.wait(lock, []{ return done; });
}

//bloqueo explicito
void custom_join2() {
    std::unique_lock<std::mutex> lock(m);
    while (!done) {
        c.wait(lock);
    }
}

/*
//bloqueo de alcance, forma invalida
void custom_join3() {
    std::lock_guard<std::mutex> lock(m);
    c.wait(m, []{ return done; });
}
//*/

int main () {
    auto child = [] () -> void {
        std::cout << "child" << std::endl;

        std::unique_lock<std::mutex> lock(m);
        done = true;
        c.notify_one();
    };

    std::thread thread(child);
    custom_join1();

    std::cout << "parent" << std::endl;
}
```

Implementation

El método `join` personalizado garantiza que el hilo principal espere hasta que el hilo hijo haya terminado su ejecución, imprimiendo siempre çhild"seguido de "parent".

3. Ejercicio 5

Revisa el ejercicio mencionado anteriormente. Ahora implementa un método join personalizado utilizando one-shot synchronization como se describe en la Sección 4.5.

`Paso 1:` Implementar un método `join` personalizado utilizando la sincronización de una sola vez

`Paso 2: Creación de la variable global y los componentes necesarios:` Primero, necesitamos una variable global `done`, un `std::promise` y un `std::future` para manejar la sincronización de una sola vez.

```
#include <iostream>
#include <thread>
#include <future>
#include <mutex>
#include <condition_variable>

bool done = false;
std::promise<void> done_promise;
std::future<void> done_future = done_promise.get_future();
std::mutex m;
std::condition_variable c;
```

`Paso 3: Definición de la función del hilo hijo`

En la función del hilo hijo, después de imprimir çhild", establecemos `done = true` y establecemos el valor de la promesa.

```
auto child = []() -> void {
    std::cout << "child" << std::endl;
    {
        std::lock_guard<std::mutex> lock(m);
        done = true;
    }
    done_promise.set_value();
};
```

`Paso 4: Implementación del método join personalizado`

El método `join` personalizado usará `done_future` para esperar hasta que el hilo hijo complete su ejecución.

```
void custom_join() {
    done_future.wait();
}
```

`Paso 5: Implementación de la función main`

En la función `main`, creamos el hilo hijo y luego llamamos a nuestro método `custom_join`.

```
int main() {
    std::thread thread(child);
    custom_join();
    std::cout << "parent" << std::endl;
    if (thread.joinable()) {
        thread.join();
    }
    return 0;
}
```

```cpp
#include <iostream>
#include <thread>
#include <future>
#include <mutex>
#include <condition_variable>

bool done = false;
std::promise<void> done_promise;
std::future<void> done_future = done_promise.get_future();
std::mutex m;
std::condition_variable c;

auto child = []() -> void {
    std::cout << "child" << std::endl;

    {
        std::lock_guard<std::mutex> lock(m);
        done = true;
    }

    done_promise.set_value();
};

void custom_join() {
    done_future.wait();
}

int main() {
    std::thread thread(child);

    custom_join();

    std::cout << "parent" << std::endl;

    if (thread.joinable()) {
        thread.join();
    }

    return 0;
}
```

Implementation

`Explicación:`

1. Definición de variables globales:
   - `done`: Booleano que indica si el hilo hijo ha terminado.
   - `done_promise` y `done_future`: Usados para la sincronización de una sola vez.
   - `m` y `c`: Mutex y variable de condición, aunque no son estrictamente necesarios para esta versión usando `promise` y `future`.
2. Función del hilo hijo:

- Imprime çhild".
- Establece `done = true` dentro de un `lock_guard` para garantizar la sincronización.
- Establece el valor de la promesa usando `done_promise.set_value()`, lo que satisface el futuro asociado.

3. Método `join` personalizado:
   - Usa `done_future.wait()` para esperar hasta que el hilo hijo complete su ejecución.
4. Función `main`:
   - Crea y lanza el hilo hijo.
   - Llama a `custom_join` para esperar a que el hilo hijo termine.
   - Imprime "parent".
   - Verifica si el hilo es joinable y llama a `thread.join()` para limpiar.

Esta implementación garantiza que siempre se imprimirá çhild"seguido de "parent", independientemente del orden de ejecución de los hilos.

4. Ejercicio 7

Recuerda la distribución cíclica de bloques programada dinámicamente para el cálculo de la matriz de distancias de todos los pares presentada en la Sección 4.4. El esquema propuesto selecciona dinámicamente bloques de tamaño fijo uno tras otro comenzando en la fila cero. Invierte el esquema de indexación de tal manera que el programa comienza con los bloques en la parte inferior de la matriz que contienen las filas más largas. ¿Observas una mejora en el tiempo de ejecución? Explica el resultado.

4.1. Declaración de Variables Globales:

```
index_t global_lower = 0;
std::mutex mutex;
```

4.2. Función Lambda para el Cálculo Dinámico:

La función lambda `dynamic_block_cyclic` se encargará de distribuir las tareas entre los hilos.

4.3. Detalles de la función `dynamic_all_pairs_rev`:

4.3.1. Parámetros

- `mnist`: Un vector que contiene los datos de las imágenes.
- `all_pair`: Un vector que almacenará las distancias de pares calculadas.
- `rows`: Número de filas (imágenes) en la matriz `mnist`.
- `cols`: Número de columnas (características o píxeles por imagen) en la matriz `mnist`.
- `num_threads`: Número de hilos a utilizar para el cálculo paralelo. Por defecto es 64.
- `chunk_size`: Tamaño del bloque de datos que cada hilo procesará. Por defecto es 64/sizeof(*value_t*).

4.3.2. Variables Globales y Bloques Cíclicos Dinámicos

- `global_lower`: Una variable compartida que mantiene el índice inferior actual que necesita ser procesado. Se utiliza para coordinar los hilos.
- `mutex`: Un mutex para proteger el acceso a `global_lower` y asegurar que solo un hilo a la vez lo modifique.

4.3.3. Lambda `dynamic_block_cyclic`

La función lambda `dynamic_block_cyclic` es la unidad de trabajo que cada hilo ejecutará. Aquí están los pasos detallados:

- **Inicialización**: `lower` se inicializa a 0, representando el índice inferior del bloque que el hilo procesará.
- **Bucle de Trabajo**: Mientras `lower` sea menor que `rows`, el hilo continuará procesando.
- **Sección Crítica**: Se utiliza un `std::lock_guard` para bloquear el acceso a `global_lower`. Dentro de la sección crítica:
  - `lower` se actualiza con el valor de `global_lower`.
  - `global_lower` se incrementa por `chunk_size` para asignar el siguiente bloque.
- **Cálculo de Límites del Bloque**: Se determina el límite superior del bloque como el menor valor entre `lower + chunk_size` y `rows`. Se calcula `LOWER` como `upper - chunk_size` si `upper` es mayor o igual a `chunk_size`, de lo contrario se asigna a 0.
- **Cálculo de Distancias**: Para cada par de imágenes (i, I) dentro del bloque:
  - Se calcula la distancia Euclidiana cuadrada entre las imágenes i e I.
  - El resultado se almacena en `all_pair` tanto en la posición $i \times \text{rows} + I$ como en $I \times \text{rows} + i$.

4.3.4. Creación y Unión de Hilos

- Se crea un vector de hilos.
- Se lanzan `num_threads` hilos, cada uno ejecutando `dynamic_block_cyclic`.
- Se espera a que todos los hilos terminen su ejecución con `thread.join()`.

4.3.5. Diferencias con `dynamic_all_pairs`

La principal diferencia entre `dynamic_all_pairs_rev` y `dynamic_all_pairs` es la forma en que se calculan los límites `lower` y `upper` para el procesamiento de bloques. En `dynamic_all_pairs_rev`, los bloques se procesan en orden inverso, desde la parte superior de la matriz hacia abajo, mientras que en `dynamic_all_pairs`, los bloques se procesan en orden natural desde la parte inferior hacia arriba. Esta variación puede ayudar en ciertos contextos de balance de carga y eficiencia de caché.

4.4. Implementación

```
void dynamic_all_pairs_rev(
        std::vector<value_t>& mnist,
        std::vector<value_t>& all_pair,
        index_t rows,
        index_t cols,
        index_t num_threads=64,
        index_t chunk_size=64/sizeof(value_t)) {

    // declare mutex and current lower index
    index_t global_lower = 0;

    auto dynamic_block_cyclic = [&] (const index_t& id ) -> void {

        // assume we have not done anything
        index_t lower = 0;

        // while there are still rows to compute
        while (lower < rows) {

            // update lower row with global lower row
            {
                std::lock_guard<std::mutex> lock_guard(mutex);
                lower  = global_lower;
                global_lower += chunk_size;
            }

            // compute the upper border of the block (exclusive)
            const index_t upper = rows  >= lower ? rows-lower : 0;
            const index_t LOWER = upper >= chunk_size ? upper-chunk_size : 0;

            // for all entries below the diagonal (i'=I)
            for (index_t i = LOWER; i < upper; i++) {
                for (index_t I = 0; I <= i; I++) {

                    // compute squared Euclidean distance
                    value_t accum = value_t(0);
                    for (index_t j = 0; j < cols; j++) {
                        value_t residue = mnist[i*cols+j]
                                        - mnist[I*cols+j];
                        accum += residue * residue;
                    }

                    // write Delta[i,i'] = Delta[i',i]
                    all_pair[i*rows+I] =
                    all_pair[I*rows+i] = accum;
                }
            }
        }
    };

    // business as usual
    std::vector<std::thread> threads;

    for (index_t id = 0; id < num_threads; id++)
        threads.emplace_back(dynamic_block_cyclic, id);
```

```
56
57      for (auto& thread : threads)
58          thread.join();
59  }
```

Implementation

4.5. Explicación de los resultados

Los resultados obtenidos muestran que la función `dynamic_all_pairs_rev` es ligeramente más rápida que la función `dynamic_all_pairs` original. Aquí está el análisis detallado:

Resultados

dynamic_all_pairs:

- Carga de datos desde el disco: 0.023642s
- Cálculo de distancias: 23.707s
- Guardado de resultados en el disco: 0.0070773s

dynamic_all_pairs_rev:

- Carga de datos desde el disco: 0.0227069s
- Cálculo de distancias: 22.859s
- Guardado de resultados en el disco: 0.0080092s

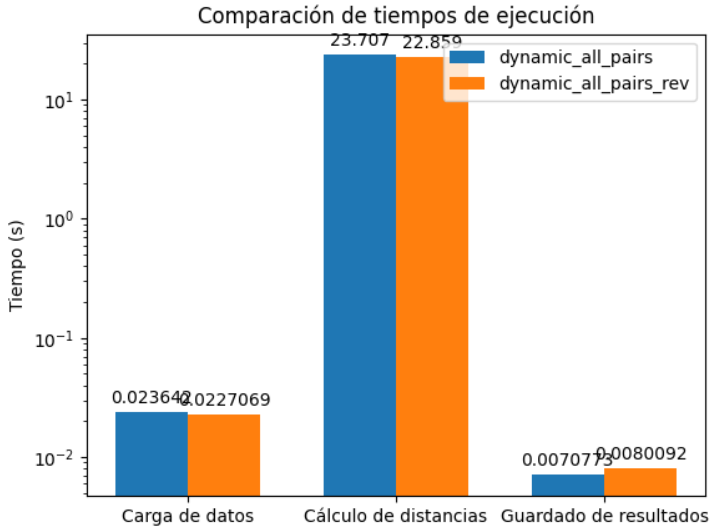

Figura 1: Grafico

Análisis del Tiempo de Ejecución

La diferencia clave se encuentra en el tiempo de cálculo de distancias:

- Tiempo de cálculo de distancias original: 23.707s
- Tiempo de cálculo de distancias invertido: 22.859s

La función `dynamic_all_pairs_rev` ha reducido el tiempo de cálculo de distancias en aproximadamente 0.848s, lo que representa una mejora de alrededor del 3.58 %.

Detalles adicionales de los resultados

- **Distribución de Carga:** Al comenzar desde la parte inferior de la matriz, donde las filas pueden tener más elementos relevantes (especialmente en matrices donde las filas inferiores tienden a ser más densas o grandes), la carga de trabajo puede distribuirse de manera más eficiente entre los hilos. Esto podría resultar en un mejor balanceo de carga y menor tiempo de espera para los hilos, ya que están procesando fragmentos más sustanciales desde el principio.
- **Localidad de Datos:** Acceder a filas consecutivas de manera invertida puede mejorar la localidad de los datos, aprovechando mejor las cachés de la CPU y reduciendo los tiempos de acceso a la memoria.
- **Optimización de la Sincronización:** Con menos hilos quedándose inactivos esperando por nuevas tareas, se puede optimizar el uso de la CPU y minimizar el overhead de sincronización.

La implementación de la función `dynamic_all_pairs_rev` ha demostrado ser más eficiente que la implementación original `dynamic_all_pairs`. Aunque la mejora en el tiempo de ejecución no es drástica, es significativa en un contexto de procesamiento paralelo y cálculo de matrices de distancias de gran tamaño.

5. link del repositorio:

GitHub.