Proyecto informatico
“Inflación Chile vs Perú”

Asignatura - Computación paralela y distribuida 2024-I

Responsables:

Diego Ruiz Suazo | 20.819.744-4
Fabián Paillacán Huaitro | 20.922.297-3

Santiago, 30 de junio de 2024

Índice de contenido

Capítulo 1: introducción

Existen fundadas dudas sobre la fiabilidad de los datos de inflación, lo cual ha causado que se lleve a cabo una comparación entre la inflación de Perú y Chile. Para lograr este objetivo, se requiere hallar la inflación de ambos países utilizando los precios de una canasta de productos. Para tal efecto, se nos han proporcionado dos archivos de datos.
El primer archivo contiene información sobre productos comprados en Perú a lo largo de cuatro años. Los datos están organizados en columnas separadas por punto y coma (“;”) y encerrados entre comillas dobles. Mientras que el segundo archivo cuenta con los precios históricos de los soles peruanos el cual ayudará a la conversión de soles a pesos chilenos.
Para lograr todo esto se planeará la elaboración de un código secuencial en el lenguaje C++, donde posteriormente se procederá a paralelizar para obtener métricas de comparación entre ambos códigos.

Capítulo 2: Planteamiento del problema

Existen fundadas dudas sobre la fiabilidad de los datos de inflación, por ende se solicita hacer una comparación de la inflación de Perú y Chile, hallando la inflación de dichos países a través de los precios de una canasta de productos, para ellos se nos da entrega de dos archivos.

Primer archivo: Contiene datos sobre los productos comprados en Perú a lo largo de 4 años. Las columnas están separadas por punto y coma (“;”) y los datos están encerrados entre comillas dobles, de la siguiente forma:

Fecha de creación: Las fechas de creación se encuentran en formato ISO 8601 extendido, es decir “2023-04-24 18:31:31.310237”, en donde la parte antes de espacio representa el año-mes-día, mientras que la parte posterior al espacio representa la hora, minutos, segundos y milisegundos.

Número de boleta: El número de la boleta se encontrará identificado por un número entero de la siguiente manera “171321”, además agrupa los productos adquiridos en una misma compra.

Número de tienda: El número de la tienda está representado por un número entero como se muestra a continuación “192”.

Nombre de fantasía de la tienda: El nombre de fantasía de la tienda está representado por un texto es decir ““MAXICOMPRA”.

Categoría de la tienda: La categoría de la tienda puede ser presentada tanto por un texto como por un número.

Tipo de envío: El tipo de envío será representado por un texto el cual indicará el formato por el cual el producto fue entregado al cliente, ejemplo ““Despacho a domicilio”.

Identificador de producto: el identificador del producto será representado por un texto alfanumérico y este representa el SKU “Stock Keeping Unit”, es decir “100101XLM”.

Cantidad: La cantidad de productos será representada por un número ejemplo “10”.

Nombre del producto: EL nombre de producto podrá ser representado tanto por texto como por formato alfanumérico ejemplo: “MabeTop Freezer RMA255FYPG 239L Grafito”

Monto (en moneda peruana): El monto será representado por numeros decimales para dar alusión a que los valores se encuentran en pesos peruanos es decir “1500.99”.

Segundo archivo: Contiene las paridades históricas entre el sol peruano y el peso chileno.

El objetivo General

Obtener una canasta básica con los productos que se repiten al menos **una vez todos los meses durante un mismo año**,utilizando como parámetro el “SKU”, todo esto a través del archivo entregado denominado “Pd.csv” el cual contiene 4 años: 2021, 2022, 2023 y 2024.

Solución propuestas

Para la resolución de la problemática planteada, se propuso realizar un código en el lenguaje de programación “C++” , esto a través del compilador denominado “DevC++”, en donde de primera instancia se abordará la problemática con la creación de un programa generado de forma secuencial, para posteriormente pasar a paralelizar dicho código con ayuda de la biblioteca denominada “OpenMP”.

Características del Ordenador

Procesador:
Familia del procesador: Intel Core i9-12900K.
Velocidad del procesador: 3.20 GHz.
Generación del procesador: 12th generación.
Memoria:
Tipo de memoria: DDR5
Ancho de bus de memoria: 64 bits
Velocidad de memoria: 4800 MHz
Disco Duro:
Tipo de disco duro: SSD
Modelo del disco duro:Kingston KC3000
Velocidad de lectura: 9166 GB
Velocidad de escritura: 6025 GB
Tamaño del caché: 45,4 MB

Capítulo 3: Elaboración del código secuencial

Antes de comenzar con la elaboración del código secuencial, es fundamental seleccionar las bibliotecas necesarias para la estructuración de este mismo:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <stdexcept>
#include <limits>
#include <algorithm>
#include <map>
#include <regex>
#include <set>
#include <utility>
#include <iomanip> // Necesario para std::fixed y std::setprecision
#include <chrono> // Necesario para medir el tiempo de ejecución

using namespace std;
using namespace std::chrono; // Para facilitar el uso de las herramientas de <chrono>

const char DELIMITADOR = ';';
const char DELIMITADOR_2 = '\t'; // Cambiamos el delimitador a tabulador
```

- <iostream> : Se utilizará para la entrada y salida estándar, como cout y cin.
- <fstream> : Esto nos servirá para la lectura de los archivos .csv y .tsv.
- <sstream> : Con esto se podrán manipular las cadenas de texto. Más adelante se verán funciones como *istringstream* y *ostringstream*.
- <vector> : Para crear nuestro vector que será contenedor para nuestros datos.
- <string> : Con esto se va a permitir usar las cadenas de texto.
- <stdexcept> : Utilizada para manejar excepciones estándar, como el argumento invalido o que está fuera de rango.
- <limits>: Utilizada para obtener las características de los tipos de datos primitivos, como *numeric_limits*.
- <algorithm> : Esta librería nos permite utilizar las funciones *sort, find* y *remove.*
- <map> : Nos permite mapear nuestro vector para encontrar repetidos. Utilizando el contenedor *“map”.*
- <regex>: Utilizada para manejar expresiones regulares, que permiten buscar y manipular texto mediante patrones.
- <set>: Sirve para manejar el contenedor *“set*”, para almacenar un solo elemento de los repetidos.
- <utility>: Utilizada para manejar utilidades generales como *pair* y *make_pair.*
- <iomanip>: La utilizamos para la salida de datos, como el *fixed* y *el setprecision*.
- <chrono>: Esta librería nos permite tomar el tiempo de ejecución del programa con *high_resolution_clock* y *duration.*

Primero que todo comenzamos con la lectura del archivo .csv (el que contiene 56 millones de datos aproximadamente). Para la primera instancia no se nos había ocurrido almacenar el archivo dentro de un vector. Hablando con unos compañeros se llegó a la conclusión de que era la mejor opción el almacenar todos los datos dentro de un vector, por otro lado también se creó la estructura, con todas las columnas que se mencionaron anteriormente, quedando de la siguiente forma:

```
struct Canasta {
    string fechaCreacion;
    int numeroBoleta;
    int numeroTienda;
    string nombreFantasia;
    string categoriaTienda;
    string tipoEnvio;
    string SKU;
    int cantidad;
    string nombreProducto;
    double monto;
};
```

Cabe destacar que para el proceso de lectura y almacenaje, se tuvo que pasar por la eliminación de comillas, formatear la fecha (dado que tenía también la hora de cuando se compró). Como también por otro lado el pasar los string a ***double*** y a ***int***.

```
string EliminarComillas(const string& str) {
    if (str.length() >= 2 && str[0] == '"' && str[str.length() - 1] == '"') {
        return str.substr(1, str.length() - 2);
    }
    return str;
}

int Cadena_entero(const string& str) {
    istringstream iss(str);
    int value;
    iss >> value;
    if (iss.fail()) {
        throw invalid_argument("No se puede convertir a entero");
    }
    return value;
}

double Cadena_double(const string& str) {
    istringstream iss(str);
    double value;
    iss >> value;
    if (iss.fail()) {
        throw invalid_argument("No se puede convertir a Double");
    }
    return value;
}

string TruncarFecha(const string& dateTime) {
    return dateTime.substr(0, 10);
}
```

Luego cuando intentamos sacar la fecha mínima y máxima del archivo, nos dimos cuenta que existían palabras en la zona donde estaban las fechas. Como así también que el archivo no se encontraba ordenado. Entonces también en el código se tuvo que poner un código que validará la fecha, es decir que los caracteres sean números y no letras.

```
bool validarFecha(const string& fecha) {
    if (fecha.size() != 10) return false;
    if (fecha[4] != '-' || fecha[7] != '-') return false;
    for (size_t i = 0; i < fecha.size(); ++i) {
        if (i == 4 || i == 7) continue;
        if (!isdigit(fecha[i])) return false;
    }
    return true;
}
```

Entonces cuando ya leímos limpiamente todo el archivo, se llamará a la función que va a ordenar nuestro archivo de forma ascendente con respecto a la fecha.

```
// Ordenar las canastas por fecha de creación
OrdenarFecha(canastas);
```

```
void OrdenarFecha(vector<Canasta>& canastas) {
  cout << "ordenando....." << endl;
  sort(canastas.begin(), canastas.end(), compararPorFecha);
}
```

Una vez ya ordenado nuestro vector, nosotros sacamos la canasta básica. Para comenzar haciendo esto, teníamos que identificar aquel producto que se repetiera todos los meses al menos una vez cada mes. Entonces nos acordamos de la función ***<map>***, el que justamente lo vimos en clases. La función agrupa los montos y la cantidad de compras de productos (identificados por su SKU) por mes para un año específico dado.

```
map<string, map<string, pair<double, int>>> agruparPorMesYProducto(const vector<Canasta>& canastas, const string& fecha) {
    map<string, map<string, pair<double, int>>> productosPorMes;
    for (const auto& canasta : canastas) {
        if (canasta.fechaCreacion.size() >= 7 && canasta.fechaCreacion.substr(0, 4) == fecha) {
            string mes = canasta.fechaCreacion.substr(0, 7); // Formato "YYYY-MM"
            productosPorMes[mes][canasta.SKU].first += canasta.monto;
            productosPorMes[mes][canasta.SKU].second += 1;
        }
    }
    return productosPorMes;
}
```

La gracia de esta función presentada es que también vamos acumulando el monto, para luego ir dividiéndolo por la cantidad de veces que se repite en el mes, sacando así el promedio.

Luego de todo esto, obteniendo los productos repetidos y sus montos totales por meses, no nos sirven que se repitan muchas veces, entonces utilizaremos la función ***<set>***, que la gracia que tiene es que no importa la cantidad de veces que se repita algo, solo guardara uno.

Entonces lo vamos a guardar en “obtenerProductosEnTodosLosMeses” , **solo uno**.

```
set<string> obtenerProductosEnTodosLosMeses(const map<string, map<string, pair<double, int>>>& productosPorMes) {
    map<string, int> contadorProductos;
    int numMeses = productosPorMes.size();

    for (const auto& entry : productosPorMes) {
        for (const auto& producto : entry.second) {
            contadorProductos[producto.first]++;
        }
    }

    set<string> productosEnTodosLosMeses;
    for (const auto& entry : contadorProductos) {
        if (entry.second == numMeses) {
            productosEnTodosLosMeses.insert(entry.first);
        }
    }
    return productosEnTodosLosMeses;
}
```

¿De qué servía todo este proceso? bueno este proceso fue llamado desde la función, “calcularInlfacion”, el cual como se expresó anteriormente, utilizamos dos funciones importantes como el ***<set>*** y ***<map>***. Todo para sacar la inflación e IPC que hay entre los meses de cada año. Por otro lado, obtendremos la canasta básica por año.

```
// Implementación de la función calcularIPCAnual como se mostró anteriormente
void calcularInflacion(const set<string>& fechas, const vector<Canasta>& canastas) {
    for (const auto& fecha : fechas) {
        auto productosPorMes = agruparPorMesYProducto(canastas, fecha);
        auto productosEnTodosLosMeses = obtenerProductosEnTodosLosMeses(productosPorMes);

        std::map<std::string, double> sumaPromediosPorMes;
        std::map<std::string, int> totalOcurrenciasPorMes;

        // Calcular sumas y promedios por mes y producto
        for (const auto& producto : productosEnTodosLosMeses) {
            for (const auto& mes : productosPorMes) {
                if (mes.second.find(producto) != mes.second.end()) {
                    double montoDelMes = mes.second.at(producto).first;
                    int cantidadEnMes = mes.second.at(producto).second;
                    double promedioPorMes = montoDelMes / cantidadEnMes;

                    sumaPromediosPorMes[mes.first] += promedioPorMes;
                    totalOcurrenciasPorMes[mes.first] += cantidadEnMes;
                }
            }
        }
```

Se puede apreciar acá en la primera mitad que utilizamos una ***const auto&*** fecha para almacenar todas las fechas, con esto almacenado nosotros podíamos dirigirnos a las funciones de ***<set>*** y de ***<map>*** (para obtener precios y productos). Entonces, ya en el segundo for, para cada producto presente en todos los meses, se iteran los meses y se calculan las sumas de los promedios por mes, así como el total de ocurrencias por mes.

```
    double totales = 0.0;
    double IPC = 0.0;
    double PrimeraVez = 0.0;
    double IPC_anterior = 0.0;
    double Inflacion = 0.0;
    int contador_IPC = 0;
    int Conta_historia = 0;
    double mes_chileno = 0.0;
    double PrimeraVez_chile = 0.0;
    double IPC_ch = 0.0;
    double Inflacion_chile = 0.0;
    double IPC_anterior_ch = 0.0;

    // Calcular IPC y mostrar resultados por mes
    for (const auto& mes : sumaPromediosPorMes) {
      mes_chileno = mes.second * precios[Conta_historia].price;
      if (mes.first.substr(0, 7) == "2021-02" ||
          mes.first.substr(0, 7) == "2022-01" ||
          mes.first.substr(0, 7) == "2023-01" ||
          mes.first.substr(0, 7) == "2024-01") {
        PrimeraVez = mes.second;
        PrimeraVez_chile = mes_chileno;
      }
      IPC_ch = (mes_chileno / PrimeraVez_chile) * 100;
      IPC = (mes.second / PrimeraVez) * 100;
      totales += mes.second;

      if (contador_IPC > 0) {
        Inflacion = ((IPC - IPC_anterior) / IPC_anterior) * 100;
        Inflacion_chile = ((IPC_ch - IPC_anterior_ch) / IPC_anterior_ch) * 100;
        std::cout << "Calculo de la inflacion Peruana: " << Inflacion
                  << " fecha: " << mes.first << std::endl;
        std::cout << "Calculo de la inflacion Chilena: " << Inflacion_chile
                  << " fecha: " << mes.first << std::endl;
      }

      contador_IPC++;
      IPC_anterior = IPC;
      IPC_anterior_ch = IPC_ch;
      Conta_historia++;
    }
  }
}
```

En esta segunda parte de la función “CalcularInflacion” se declaran muchas variables que servirán para calcular la inflación chilena y peruana, así también el IPC que justamente sirve para la inflación.

```cpp
void calcularPromedioPreciosPorMes(vector<PreciosHistoricos>& precios) {
  map<pair<int, int>, pair<double, int>>
      promedios;  // (año, mes) -> (suma de precios, cantidad de elementos)

  // Calcular promedios por mes y año
  for (const auto& precio : precios) {
    // Obtener año y mes de la fechaPrecio
    size_t pos = precio.fechaPrecio.find('/');
    if (pos != string::npos) {
      int day = stoi(precio.fechaPrecio.substr(0, pos));
      size_t pos2 = precio.fechaPrecio.find('/', pos + 1);
      if (pos2 != string::npos) {
        int month = stoi(precio.fechaPrecio.substr(pos + 1, pos2 - pos - 1));
        int year = stoi(precio.fechaPrecio.substr(pos2 + 1));
        // Validar rango de fechas nuevamente (aunque debería estar validado ya
        // en leerPreciosHistoricos)
        if (year >= 2021 && year <= 2024 && month >= 1 && month <= 12 &&
            day >= 1 && day <= 31) {
          promedios[{year, month}].first += precio.price;
          promedios[{year, month}].second++;
        }
      }
    }
  }

  // Limpiar el vector precios
  precios.clear();

  // Actualizar el vector con los promedios calculados
  for (const auto& entry : promedios) {
    auto key = entry.first;
    auto value = entry.second;
    double promedio = value.first / value.second;
    precios.push_back(
        {to_string(key.second) + "/" + to_string(key.first), promedio});
  }
}
```

En esta función lo que se logra, es sacar los promedios de la suma de precios que hay entre todos los SKU de productos. Primero se crea un contenedor tipo ***<map>*** Este mapa se utilizará para acumular los precios y contar el número de elementos por cada mes y año.

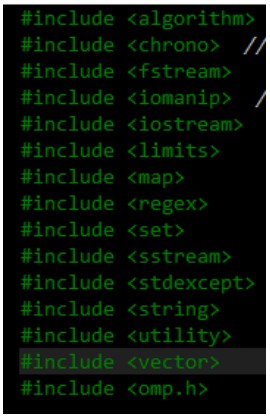
Entonces vamos iterando en el vector precios para ir actualizando el mapa promedios, sumando el precio al acumulado correspondiente al (año, mes) y aumentando el contador de elementos. Después casi al final limpiamos nuestro vector <precios> con un .clear(). Todo para rellenarlo nuevamente con los promedios calculados, lo que se realiza en el último for.

```cpp
int main() {
    // Inicio de la medición de tiempo
    auto start = high_resolution_clock::now();
    set<string> fechas = {"2021", "2022", "2023", "2024"};
    vector<Canasta> canastas;

    // Leer el archivo y llenar el vector de canastas
    LeerArchivo(canastas);

    // Leer precios históricos
    vector<PreciosHistoricos> precios;
    leerPreciosHistoricos(precios);

    // Calcular promedio de precios por mes y año
    calcularPromedioPreciosPorMes(precios);

    calcularInflacion(fechas, canastas, precios);

    // Fin de la medición de tiempo
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(end - start);

    cout << "Tiempo de ejecución: " << duration.count() << " ms" << endl;
    return 0;
}
```

En este punto con el programa terminado, la ejecución del mismo demoraba 10.4 minutos. Entonces para finalizar nos dimos cuenta que el ***<map>*** no necesita trabajar con el archivo ordenado, entonces después cuando borramos las funciones que tenían que ver con ordenar, nos demoramos 5.4 minutos. Para finalizar entonces, el código secuencial no cuenta con la función ordenar fechas, por lo tanto el tiempo de ejecución del código secuencial es de 5.4 minutos.

Capítulo 4: Elaboración del código paralelo

Para la elaboración del código paralelo lo primero que se realizó fue la declaración e inclusión de la biblioteca llamada “omp.h”, la cual nos brindará las herramientas necesarias para paralizar el código de la forma más eficiente posible.



Posteriormente se procedió a eliminar la función “OrdenarFecha”, esto a que lo único que hacía era consumir tiempo de ejecución del programa. Nosotros declaramos una función ***<map>*** la cual es la encargada de obtener la canasta de cada año y esta función no necesita que el vector “canastas” este ordenador para funcionar efectivamente. Nos logramos dar cuenta de esto debido a que se realizó una investigación de cómo era el funcionamiento de la función ***<map>*** y poniendo a prueba varios mini programas con la función ***<map>***.

```
// Ordenar las canastas por fecha de creación
OrdenarFecha(canastas);
```

```
void OrdenarFecha(vector<Canasta>& canastas) {
  cout << "ordenando....." << endl;
  sort(canastas.begin(), canastas.end(), compararPorFecha);
}
```

Una vez eliminada la función “OrdenarFecha”, nos propusimos paralelizar la función “leerArchivo”, pero nos percatamos que esto es ineficiente en nuestro caso, debido a que el archivo .csv entregado debe leerse linea por linea, para esto solo podremos dejar que un solo hilo haga manejo de dicha función, aparte que según investigaciones propias, cuando tenemos un while que tiene funciones ***try*** y ***catch***, solo debe ser manejada por un solo hilo, debido a que si se intentara manejar la función con más de un hilo, se provocaría una condición de carrera en donde los hilos competirán por los recursos del programa, por ende se buscó paralelizar el ***<map>*** “agruparPorMesYProducto”, en donde esté ***<map>*** busca obtener todos los productos que se repiten en todos los meses de un mismo año. El primer cambio que se hizo con respecto al código secuencial fue cambiar la forma de “for”, para que pueda ser compatible con el “pragma omp parallel for”, además se indicó que solo se utilizaran 12 hilos, debido a que ya sea que se utilicen más o menos hilos en tiempo de ejecución empeorara.

```
map<string, map<string, pair<double, int>>> agruparPorMesYProducto(
    const vector<Canasta>& canastas, const string& fecha) {
  map<string, map<string, pair<double, int>>> productosPorMes;
  #pragma omp parallel for num_threads(12)
  for (int i = 0; i < canastas.size(); ++i) {

    if (canastas[i].fechaCreacion.size() >= 7 &&
        canastas[i].fechaCreacion.substr(0, 4) == fecha) {
      string mes = canastas[i].fechaCreacion.substr(0, 7);  // Formato "YYYY-MM"
      productosPorMes[mes][canastas[i].SKU].first += canastas[i].monto;
      productosPorMes[mes][canastas[i].SKU].second += 1;
    }
  }
  return productosPorMes;
}
```

Por último se buscó paralelizar la función “calcularInflacion”, la cual tiene como propósito calcular la inflación de Perú y Chile, esto a través del vector “precios” y “canastas”, en donde de primera instancia se cambia la forma de llamar a los “for”, para tener completa compatibilidad con el manejo del “Pragma omp parallel for”. En esta función se indica que solo se trabajara con 12 hilos, a su vez se incorporó el uso de “Pragma omp atomic”, esto porque se busca evitar que se genere una condición de carrera entre los diversos hilos que están modificando a la variable y de esta manera impidiendo que generen valores erróneos.

```
//-------------------------------------------------------------------------------
// Implementación de la función calcularIPCAnual como se mostró anteriormente
void calcularInflacion(const set<string>& fechas,
                       const vector<Canasta>& canastas,
                       const vector<PreciosHistoricos>& precios) {
   #pragma omp parallel for  num_threads(12)
  for (int i = 0; i < fechas.size(); ++i) {
    auto fecha = fechas.begin();
    advance(fecha,i);
    auto productosPorMes = agruparPorMesYProducto(canastas, *fecha);
    auto productosEnTodosLosMeses =
        obtenerProductosEnTodosLosMeses(productosPorMes);

    std::map<std::string, double> sumaPromediosPorMes;
    std::map<std::string, int> totalOcurrenciasPorMes;

    // Calcular sumas y promedios por mes y producto

    for (int z = 0; z < productosEnTodosLosMeses.size(); ++z) {
        auto producto = productosEnTodosLosMeses.begin();
    advance(producto ,z);
      for (const auto& mes : productosPorMes) {
        if (mes.second.find(*producto) != mes.second.end()) {
          double montoDelMes = mes.second.at(*producto).first;
          int cantidadEnMes = mes.second.at(*producto).second;
          double promedioPorMes = montoDelMes / cantidadEnMes;
        #pragma omp atomic
          sumaPromediosPorMes[mes.first] += promedioPorMes;
          #pragma omp atomic
          totalOcurrenciasPorMes[mes.first] += cantidadEnMes;
        }
      }
    }
```

```
    double totales = 0.0;
    double IPC = 0.0;
    double PrimeraVez = 0.0;
    double IPC_anterior = 0.0;
    double Inflacion = 0.0;
    int contador_IPC = 0;
    int Conta_historia = 0;
    double mes_chileno = 0.0;
    double PrimeraVez_chile = 0.0;
    double IPC_ch = 0.0;
    double Inflacion_chile = 0.0;
    double IPC_anterior_ch = 0.0;

    // Calcular IPC y mostrar resultados por mes
    #pragma omp parallel for  num_threads(12)
    for (int y = 0; y < sumaPromediosPorMes.size(); ++y) {
            auto mes = sumaPromediosPorMes.begin();
    advance(mes,y);
      mes_chileno = mes->second * precios[Conta_historia].price;
      if (mes->first.substr(0, 7) == "2021-02" ||
          mes->first.substr(0, 7) == "2022-01" ||
          mes->first.substr(0, 7) == "2023-01" ||
          mes->first.substr(0, 7) == "2024-01") {
        PrimeraVez = mes->second;
        PrimeraVez_chile = mes_chileno;
      }
      IPC_ch = (mes_chileno / PrimeraVez_chile) * 100;
      IPC = (mes->second / PrimeraVez) * 100;
      totales += mes->second;
      if (contador_IPC > 0) {
        Inflacion = ((IPC - IPC_anterior) / IPC_anterior) * 100;
        Inflacion_chile = ((IPC_ch - IPC_anterior_ch) / IPC_anterior_ch) * 100;
        std::cout << "Calculo de la inflacion Peruana: " << Inflacion
                  << " fecha: " << mes->first << std::endl;
        std::cout << "Calculo de la inflacion Chilena: " << Inflacion_chile
                  << " fecha: " << mes->first << std::endl;
      }
      contador_IPC++;
      IPC_anterior = IPC;
      IPC_anterior_ch = IPC_ch;
      Conta_historia++;
    }
  }
}
```

También se buscó aprovechar las funciones de la biblioteca “omp.h” para la obtención de tiempo del programa en segundos, con ayuda de “omp_get_wtime()”

```
int main() {
  // Inicio de la medición de tiempo
  double start_time = omp_get_wtime();
  set<string> fechas = {"2021", "2022", "2023", "2024"};
  vector<Canasta> canastas;

  // Leer el archivo y llenar el vector de canastas
  LeerArchivo(canastas);

  // Leer precios históricos
  vector<PreciosHistoricos> precios;
  leerPreciosHistoricos(precios);

  // Calcular promedio de precios por mes y año
  calcularPromedioPreciosPorMes(precios);

  calcularInflacion(fechas, canastas, precios);

    // Fin de la medición de tiempo
  double end_time = omp_get_wtime();
  double elapsed_time = end_time - start_time;

  cout << "Tiempo de ejecución: " << elapsed_time << " segundos" << endl;
  return 0;
}
```

Métricas de performance

Para obtener las métricas de performance entre el código paralelo y el secuencial, primero se debido obtener el tiempo de demora que tienen ambos códigos como se muestra a continuación:

Primero tenemos el tiempo de ejecución del código secuencial en donde se demoró un total de 326.799 milisegundos que equivalen a 5,4 minutos.

```
Tiempo de ejecuci¾n: 326799 ms
```

Posteriormente tenemos el tiempo de ejecución del código paralelo el cual se demora un total de 289,906 segundos que equivale a 4,8 minutos.

```
Tiempo de ejecucion paralelo: 289.906 segundos
```

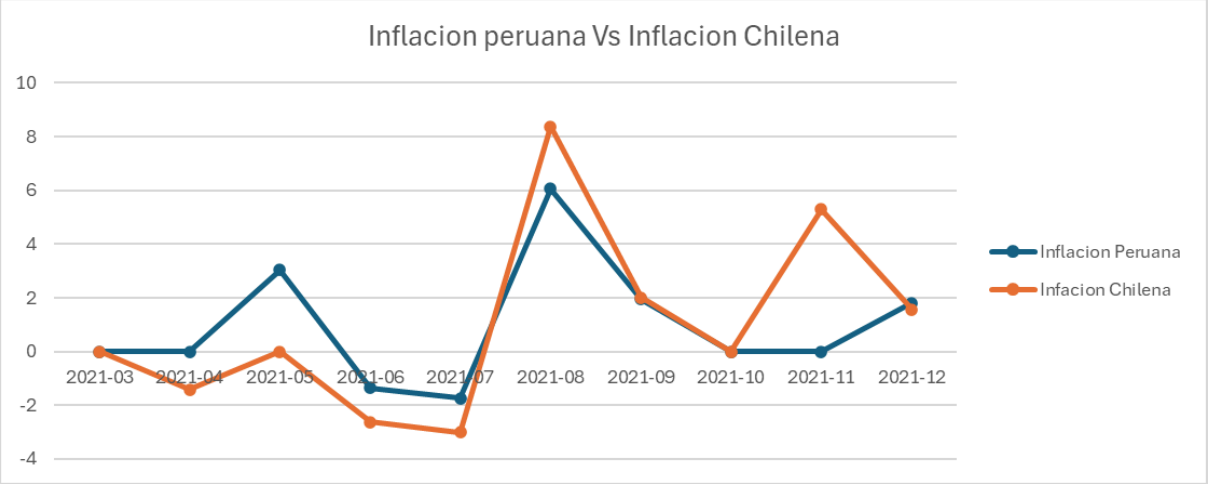
Una vez obtenido el tiempo de demora de ambos códigos, se procederá a calcular el Speedup el cual es igual a decir (Código secuencial / código Paralelo).
Speedup = 5,4/4,8 =1,125, esto significa que el programa paralelo es 1,125 más rápido que el secuencial.
Por otra parte también se calculará la ley de Amdahl que tiene el programa paralelo sobre el secuencial, para ello necesitamos tener el Speedup y el número de hilos utilizados. En donde para obtener la ley se utilizó la siguiente fórmula:
Amdahl = 1/(1-P) + (P/S) = 1/(1-P)+(P/12) = 1,125 = 0,12 *100 = 12%
Es decir que el programa se logró paralelizar un 12%, mientras que el otro 88% no logró ser paralelizado.

Capítulo 5: Comparativas de la inflación

Debido a que el archivo entregado “pd.csv”, cuenta con demasiados productos y además los parámetro utilizados para la obtención de la canasta básica de cada año era demasiado generales (esto genera que los montos por mes sean elevados), la inflación calculada difiere demasiado con la inflación real de cada país. A continuación mostraremos un gráfico de la comparativa de las inflaciones de cada país por año. (2021, 2022, 2023 y 2024)

Año 2021

Inflacion Peruana	Infacion Chilena	Fechas
0.177411	-0.568109	2021-03
-0.229933	-1,40775	2021-04
3,03654	0.699275	2021-05
-1,34676	-2,63078	2021-06
-1,74739	-3,01597	2021-07
6,06531	8,37626	2021-08
1,95826	2,00965	2021-09
0.864042	0.988058	2021-10
-0.72796	5,30270	2021-11
1,80992	1,56791	2021-12

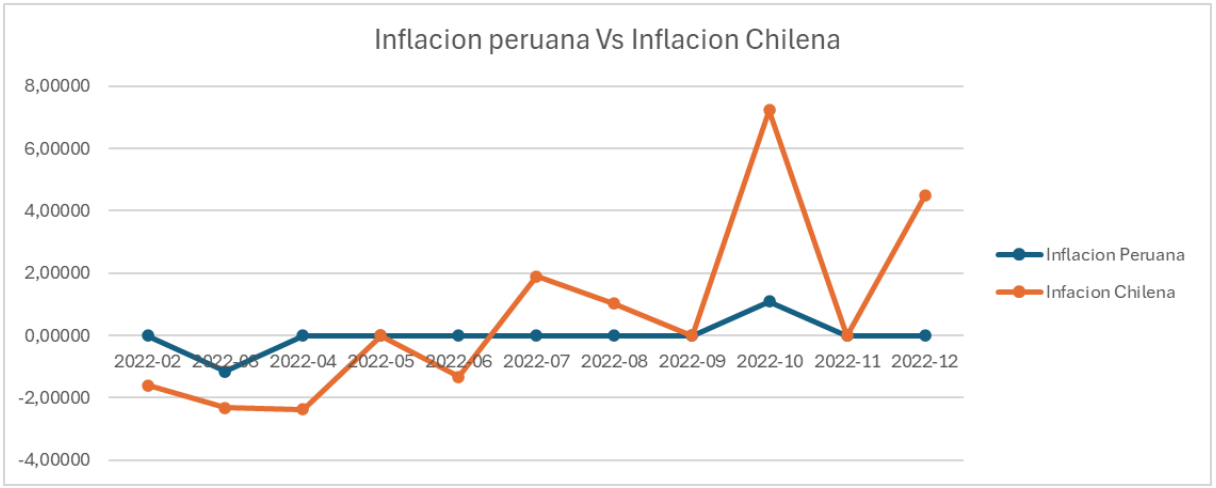


Inflación promedio en Perú: 0.986%
Desviación estándar en Perú: 2.348%
Inflación promedio en Chile: 1.132%
Desviación estándar en Chile: 3.524%

En el año 2021, se puede apreciar que tanto Chile y Perú tienen deflación en un cierto periodo de meses, para luego seguidamente obtener una alta inflación. Estos datos son comparados con la canasta del pd.csv y precios históricos.csv (o .tsv)

Año 2022

Inflacion Peruana	Infacion Chilena	Fechas
-0.855454	-1,59329	2022-02
-1,15711	-2,32398	2022-03
-0.0991519	-2,36529	2022-04
0.754229	-0.557136	2022-05
-0.0299206	-1,32068	2022-06
-0.283001	1,88963	2022-07
0.965832	1,01672	2022-08
0.76037	0.884258	2022-09
1,08901	7,23005	2022-10
-0.0701937	-0.307742	2022-11
0.476487	4,48969	2022-12

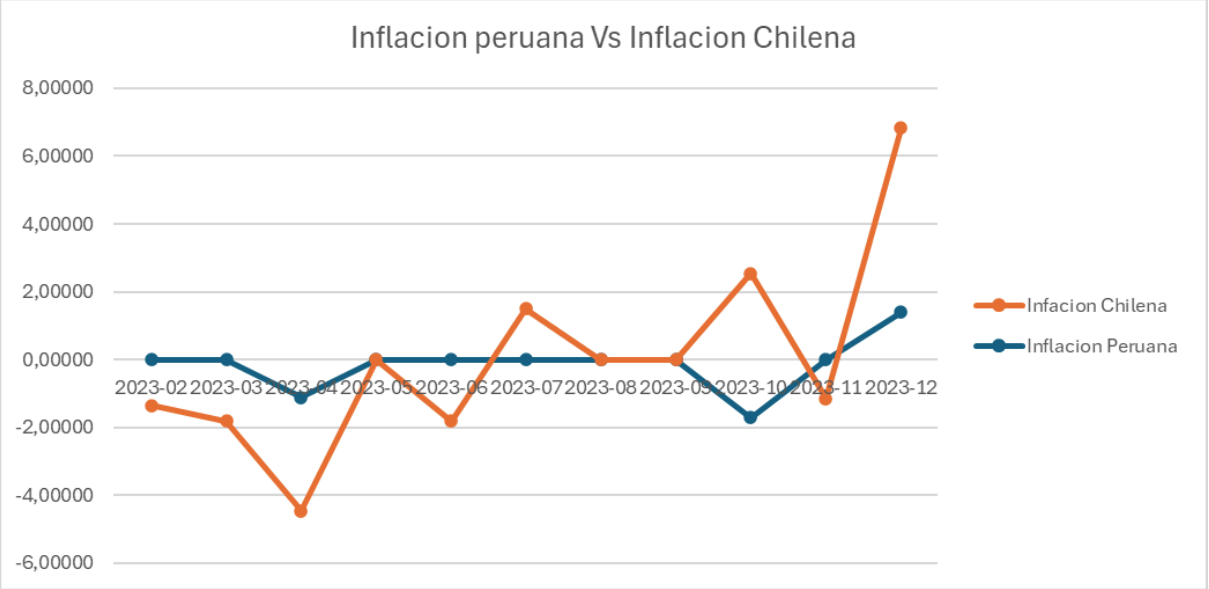


Inflación promedio en Perú: 0.1401
Desviación estándar en Perú: 0.70327
Inflación promedio en Chile: 0.54929
Desviación estándar en Chile: 2.84564

Durante el año 2022 la inflación peruana se mantuvo bastante estable, al contrario de la chilena, que tuvo fuertes variaciones dentro de los meses correspondientes del año 2022.

Año 2023

Inflacion Peruana	Infacion Chilena	Fechas
-0.607374	-1,34705	2023-02
-0.637576	-1,81058	2023-03
-1,10778	-3,35103	2023-04
0.697372	-0.613253	2023-05
-0.529713	-1,81402	2023-06
-0.665495	1,49880	2023-07
0.205155	0.255658	2023-08
0.554249	0.677883	2023-09
-1,71109	4,25984	2023-10
-0.913003	-1,14855	2023-11
1,39171	5,44147	2023-12

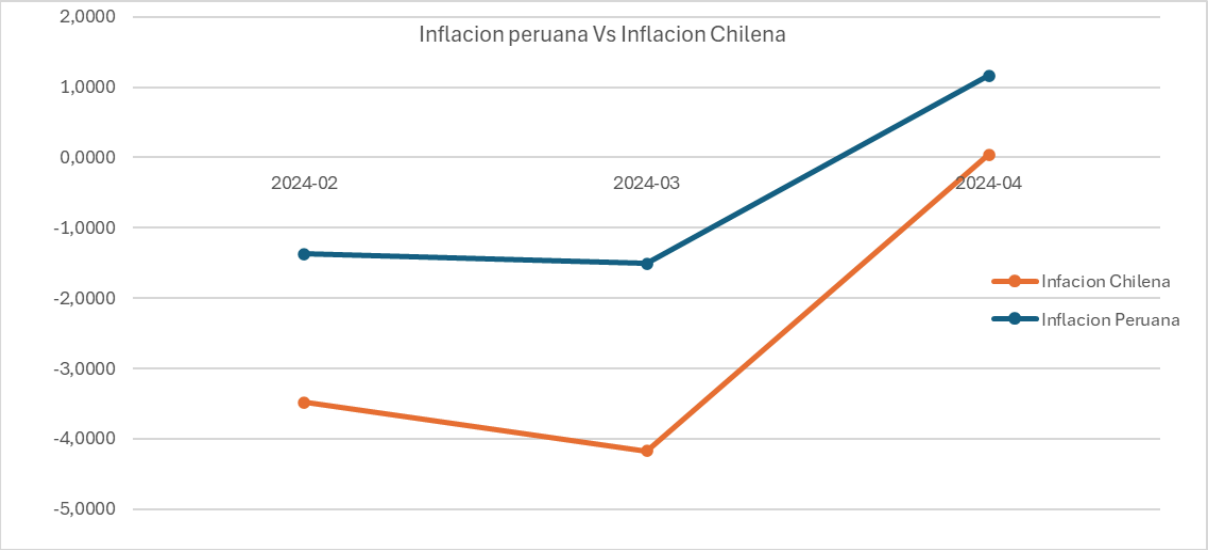


Inflación promedio en Perú: -0.196
Desviación estándar en Perú: 1.02
Inflación promedio en Chile: 0.288
Desviación estándar en Chile:2.54

La inflación peruana fue bastante similar al año anterior, salvo que llegando a septiembre se desequilibró la economía, provocando inflación y deflación. Por otro lado, en Chile, como se ha venido repitiendo todos los años, hay fuertes variaciones en el transcurso del año con respecto a la economía.

Año 2024

Inflacion Peruana	Infacion Chilena	Fechas
-1,3758	-2,1098	2024-02
-1,5059	-2,6686	2024-03
1,1667	-1,1282	2024-04



Inflación promedio en Perú: -0.57166
Desviación estándar en Perú: 0.79023
Inflación promedio en Chile: -1.96885
Desviación estándar en Chile: 0.63663

El año 2024 no arroja mucha información sobre inflaciones, dado que son solo 3 meses. Como se ve en el gráfico, ambos países empezaron con una deflación al principio del año, con respecto al mes 4, Chile llegó al punto de tener 0% de inflación y Perú cerca del 1% de inflación. Esto quiere decir que ambos países superaron la deflación con lo que va el año.

Capítulo 6: Conclusión

El presente trabajo se pudo realizar de una manera correcta, al principio de todo no teníamos muy claro cómo comenzar ni las funciones que debíamos implementar. El tiempo de la ejecucion del codigo secuencial e habia visto afectado por la función ordenamiento que teníamos, que era totalmente innecesario, nosotros no sabiamos como funcionaba bien el map, es decir, no sabiamos que podia actuar en un conjunto no ordenado y almacenarlo de forma ordenada. Por otro lado tuvimos percances a la hora de paralelizar, en primera instancia no sabíamos cuáles paralelizar y cuáles no. Intentamos mucho paralelizar la función LeerArchivo (la de 56 millones de datos), lamentablemente no pudimos dado que se nos hacían condiciones de carrera, aun cuando pusimos pragma critical y atomic. Entonces LeerArchivo lo dejamos secuencial y transformamos a ejecución paralela funciones de cálculo, la función***<set>*** y la función ***<map>*** . La percepción con la que nos quedamos es de un buen desarrollo de trabajo, sentimos que el tiempo logrado es bastante bueno, se acerca bastante a la pauta.

Capítulo 7: Bibliografía

- https://www.openmp.org/wp-content/uploads/openmp-examples-5.2.2-final.pdf
- http://lcomp89.fi-b.unam.mx/licad/assets/ProgramacionOpenMP/ProgramaciÃ³nParalelaOpenMP.pdf
- https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf