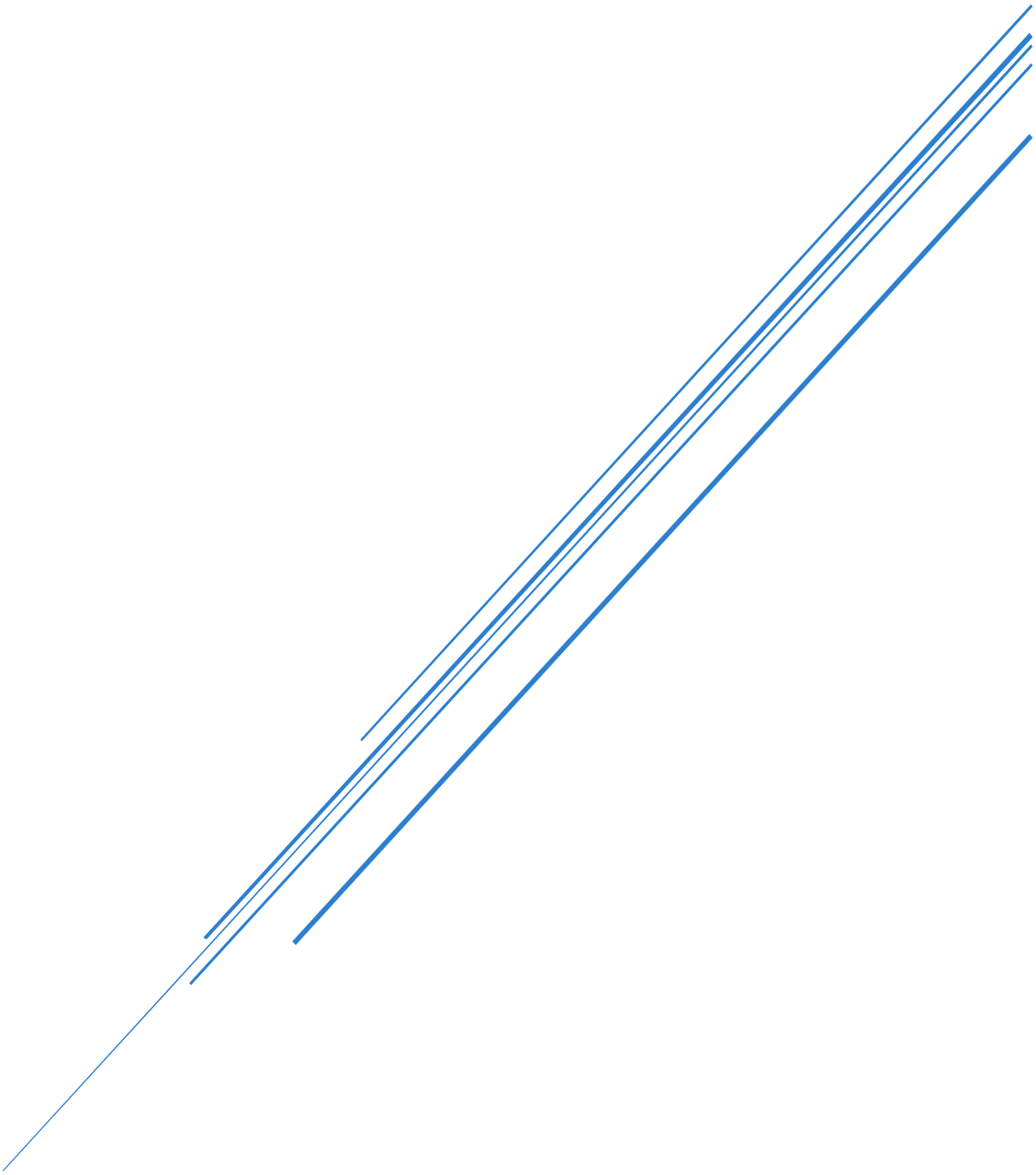
DOCUMENTACIÓN PREDICTOR CLIMÁTICO MEDIANTE ARIMA

Adrián Garrido Beas



Universidad de Murcia
Programación de Arquitecturas Multinúcleo

índice

Introducción

A continuación, procederemos a la explicación del funcionamiento, utilización y estudio del código desarrollado para el proyecto de la asignatura de programación de arquitecturas multinúcleo, dicho esto el programa consiste en dos procesos principalmente captura de datos desde un fichero JSON y análisis y cálculo de predicción de valores a razón de una continuación de valores históricos procedentes del JSON antes analizado.

Se trata de un código buscando la máxima genericidad posible es por ello por lo que se puede garantizar la reutilización de las funciones y mecanismos de este; lo cual es ventajoso, ya que la utilización de modelos como es el de ARIMA tiene gran repercusión en todos los campos, ya que este no solo se utiliza en predicción climática este con los parámetros correctos es ampliamente utilizado en economía, biología …. Se trata de un modelo que permite el análisis y predicción de entornos complejos como pueden ser colonias de seres vivos, varianzas en la bolsa de una empresa y como no estudio de sistemas climáticos basándonos en interacciones pasadas.

Por eso para realizar esta documentación segmentaré el código a razón de los módulos que lo componen.

Componentes Comunes

ARIMA

Este es el módulo de cálculo en él encontramos las diversas funciones que nos son necesarias para poder hacer el cálculo de los coeficientes que nos darán acceso al ajuste para el valor futuro.

Este veremos que se encuentra separado en un “.h” donde haremos la definición de nuestras funciones además de la estructura que formara a nuestro modelo de ARIMA.

Por ello empezando con este fragmento del código y yendo de arriba abajo veremos primero que nada el “struct” o estructura que nos muestra 3 valores del tipo entero bajo los nombres p, d, q los cuales tienen el siguiente significado:

Componente AR (p) - Autorregresivo:

Este representa cuantos valores anteriores serán tomados para realizar la siguiente predicción, viene funcionando como el proceso de entrenamiento de una IA no capturar ningún valor lleva a que las predicciones sean aleatorias en exceso, el rango entre 0 y 2 está en el punto correcto para la mayoría de procesos, ya que no es altamente agresivo siendo de hecho el ideal para las predicciones climáticas el 1, ya que si bien a razón de lo que haya sucedido antes podemos intuir como se comportara después no podemos confiar mucho en ello porque es un sistema bastante caótico, en cambio, campos como el análisis

de flujos de bolsas se utiliza el 2 o incluso en casos de empresas muy estables 3 porque esto supone series consideradas estacionarias o de poca variabilidad. Más allá de tres se puede considerar "Over fitting" o sobre ajuste por lo que las predicciones que nos del sistema no esperaran posibles cambios bruscos, ahí el símil con el entrenamiento de una IA, presentarle escenarios hace que sea capaz de analizar correctamente, pero la sobreexposición a muestras lo hace poco flexible a ejemplos externos.

Componente I (d) - Integración:

Este nos mostrará que tan a menudo se diferencia la serie entre sí misma este campo solo puede tomar 3 valores siendo el 0 que no aplica ningún tipo de diferenciación como se da el caso para series como la climática que se considera estacionaria, ya que se mantiene relativamente estable durante largos periodos de tiempo, siempre que no hablemos de los problemáticos que está siendo el clima en los últimos meses; con valor 1 eliminamos una tendencia lineal siendo el caso del estudio de sistemas que tengan un crecimiento continuo y por último la diferenciación cuadrática para sistemas que sean muy inestables esto se hará con el valor 2.

Componente MA (q) - Media Móvil:

Por último, este valor nos ayuda a ajustar mediante decidir cuanto error estamos asumiendo en el cálculo de los valores, siendo 1 y 2 tomar únicamente el valor de los respectivos valores anteriores y por último 3 o más para modelo sistemas con shocks temporales que tiendan a ser muy ruidosos

Continuando con el código en el módulo ARIMA tenemos 2 funciones denominadas como básicas que se encargan del cálculo de la media y de una regresión lineal estándar.

Siendo el caso de la función que calcula la media la entramos con su nombre en inglés "mean" esta recibirá el array de enteros que hemos capturado previamente y calculará la media de los valores tomados. Valor que será importante para la regresión lineal posterior porque el uso de la función "mean" se da dentro tanto de "forecast_arima" como de "lineal_regression"

Siendo "lineal_regresion" una función que a partir de la media de los valores del array de tiempo y de los datos capturados iterando en un bucle for sobre el espacio de los datos se calcula las diferencias entre las medias y el valor del paso en el que nos encontramos para posteriormente haciendo el sumatorio, obtendremos los valores "b0" y "b1" los cuales compuestos en el caso de "b0" por $0 = media_{valores} - \left(b1 * media_{tiempo}\right)$ y "b1" por $b1 = \frac{\sum media_{tiempo} * (valor - media_{valores})}{\sum media_{tiempo}^2}$

El cálculo de esta regresión lineal nos servirá para estimar que tan correcta sería la predicción de nuestro cálculo de ARIMA, ya que como mencione antes no solo podemos tener errores en el cálculo por la implementación; sino que además podemos ver errores por haber escogido de manera incorrecta los parámetros de ejecución.

Para la ejecución de ARIMA como modelo dependemos además de que de la media básica que estamos calculando las siguientes 3 funciones:

Difference:

Está transforma en estacionaria la secuencia de datos aplicando el criterio de diferenciación como la resta iterativa del valor i de la serie menos el i menos criterio de diferenciación:

$$diff\_series\ [i - d] = series[i] - series\ [i - d]$$

siendo d el criterio de diferenciación

esta primera función es el paso de partida para la siguiente que tiene la labor de realizar un ajuste de los datos para nuestro modelo de predicción, de nuevo partiendo de nuestra secuencia de datos se realiza la diferenciación para posteriormente aplicarle esta fórmula a la serie diferenciada.

$$params[i] = 0.5f * (i + 1) / (model->p + model->q)$$

Tras el proceso de ajuste de datos realizaremos la función de ARIMA tal que se verá el sumatorio de los productos de los coeficientes de autorregresión por los valores de la serie ajustada.

Por último el proceso realiza producto del parámetro del valor en el índice p, siendo p el componente autorregresivo previamente explicado, por lo que se considera la media móvil de la serie.

JSONParser

Como en el apartado anterior trataré los puntos en común entre ambas versiones, es por ello por lo que referenciaré las funciones, pero no el uso de MPI ni OpenMP en este apartado; de eso me encargaré posteriormente, ya que considero que se ha de tratar de forma abstraída con respecto a la funcionalidad general.

Primero que nada, veremos el struct CampoDatos. Este nos permite almacenar los datos de cada campo que queramos capturar de manera individual; con ello me refiero a que más tarde crearemos un array del struct, de manera que a través de su valor nombre identificaremos qué valor estamos extrayendo del JSON, seguido por los valores, que es un array de enteros que contendrá todos y cada uno de los valores obtenidos, y por último un entero que nos indica cuántos datos de ese tipo hemos extraído. De esa manera podemos asegurar que, a pesar de que en el cálculo general de registros en el JSON no haya la posibilidad de que un campo tenga menos valores que registros, tiene todo el fichero.

Siguiendo en orden de aparición, sabemos que en muchos casos los valores enteros no son guardados en el JSON con la estructura anglosajona “0.0”, que es la que sí es capaz de tolerar C como un valor entero. Es por ello por lo que dependemos de una función que itere sobre todo el fichero cambiando todas las “,” por “.”.

A continuación, vemos la función cuyo nombre es “array_auxiliar_dias”, debido a que para realizar la regresión lineal básica necesitamos un valor independiente, en nuestro caso los días, que en realidad será un array cuyos valores serán iguales a la posición con respecto al total de registros capturados, como sería el caso de i [100] = 100.0. A pesar de considerarse innecesario que esos sean enteros, por motivos de compatibilidad posterior los guardaremos de esa manera.

Y como función principal y más importante tendremos “extraer_campos”; esta comenzará creando un array de la estructura contenedora de nuestros datos extraídos, creará dicho array con el tamaño respectivo al número de datos que ha de buscar según le hemos pedido, seguido por un malloc equivalente a la cantidad total de registros que tenga nuestro JSON en el tamaño de un número entero.

Una vez tenemos preparada la memoria para poder contener nuestros valores, iteraremos sobre nuestro fichero JSON. Con un bucle for buscaremos aquellas entradas ítems cuyo valor, es decir, su nombre de campo coincida para ser guardado en el array valores de nuestra estructura. En caso de ser una cadena o un número con un formato incorrecto, procedemos a realizarle ajustes de formato y seguidamente guardarlos en nuestra estructura.

En este fichero también encontraremos un main, que tendrá que ser explicado con mayor profundidad en este documento, siendo el caso de que ambas versiones del programa disponen de un main distinto. Pero principalmente en ambos se llama a las funciones antes mencionadas, tanto las del cálculo como las de obtención de datos.

Hibrido OMP MPI

ARIMA

En este fragmento de código, todo funciona tal y como se describió previamente; es por ello por lo que creo que, salvo mostrar el código, no hace falta una explicación mucho más lejos de lo antes mencionado.

```c
/*
Descripción:
    -Funcion destinada a calcular la media de los valores pasados en un array.
Parametros:
    -float *array: Suministro de datos.
    -int n: Tamaño del array.
*/
float mean(float *array, int n) {
    float sum = 0.0f;
    #pragma omp parallel for reduction(+:sum) schedule(static, 8)
    for (int i = 0; i < n; i++) {
        sum += array[i];
    }
    return sum / n;
}
```

Función mean

Comenzando por la función “mean” esta calcula la media de los valores que se le pasan por un array, lo calcula en un bucle for que se encuentra paralelizado en bloque de 8 elementos garantizando que en el uso de un compilador que no vectorice de forma nativa se asegure la localidad espacial de los datos pasados.

```c
/*
Descripción:
    -Funcion que realiza la Regresion lineal
Parametros:
    -float *x: Array de valores independientes ('Dias')
    -float *y: Array de valores dependientes ('Extraido JSON')
    -int n: Tamaño de ambos arrays.
    -float *b1: Valor calculado usado para la predicción.
    -float *b0: Valor calculado usado para la predicción.
*/
void linear_regression(float *x, float *y, int n, float *b1, float *b0) {
    float mean_x = mean(x, n);
    float mean_y = mean(y, n);
    float numerator = 0.0f, denominator = 0.0f;

    #pragma omp parallel for reduction(+:numerator, denominator) schedule(static, 8)
    for (int i = 0; i < n; i++) {
        float x_diff = x[i] - mean_x;
        numerator += x_diff * (y[i] - mean_y);
        denominator += x_diff * x_diff;
    }

    *b1 = numerator / denominator;
    *b0 = mean_y - (*b1 * mean_x);
}
```

Función Linear_regression

Calculando como se mencionó antes esta también hace por aprovechar la localidad espacial de los valores con los que trabajamos tomando los datos de 8 en 8 para la caché de cada hilo mejorando ligeramente el rendimiento.

```c
/*
Descripcion:
    -Funcion para la diferenciacion o alisado de los datos en busqueda de generar series estacionarias
Parametros:
    -float *series: Array con los valores a normalizar
    -int n: Tamaño del array original
    -int d: Factor de diferenciación.
    -float *diff_series: Array contenedor de la nueva serie estilizada.
*/
void difference(float *series, int n, int d, float *diff_series) {
    #pragma omp parallel for schedule(static, 8)
    for (int i = d; i < n; i++) {
        diff_series[i - d] = series[i] - series[i - d];
    }
}
```

Función Difference

Esta función es un estabilizador de datos, ARIMA no solo se utiliza en series estacionarás como son las climáticas, sino que también encuentra series bastante más erráticas como son las series que componen los movimientos en bolsa de una empresa es por ello por lo que cuenta con el parámetro d, de diferenciación que sirve para estabilizar las series que tratamos.

```c
/*
Descripcion:
    -Funcion para el ajuste de los datos, tras la diferenciacion de los mismo.
Parametros:
    -float *series: Array de datos previamente diferenciados.
    -int n: Tamaño de la serie tras la diferenciación.
    -ARIMA_Model *model: Modelo ARIMA que estamos usando
    -float *params: Array contenedor de la serie ajustada
*/
void fit_arima(float *series, int n, ARIMA_Model *model, float *params) {
    float *diff_series = malloc((n - model->d) * sizeof(float));
    difference(series, n, model->d, diff_series);

    #pragma omp parallel for
    for (int i = 0; i < model->p + model->q; i++) {
        params[i] = 0.5f * (i + 1) / (model->p + model->q);
    }

    free(diff_series);
}
```

Función fit_arima

Esta se usa para ajustar los datos recibidos desde el JSON alisando un poco los datos porque si bien de nuevo las series climáticas no son tan extremas este software está diseñado para que pueda ser reutilizado para cualquier tipo de campo que pueda hacer uso de este tipo de cálculos estadísticos.

```c
/*
Descripción:
    -Función que calcula los valores autoregresivos y la media movil valores necesarios para la predicción del siguiente valor.
Parametros:
    -float *series: array pre estilizado
    -int n: Tamaño del array original
    -ARIMA_Model *model: Modelo de ARIMA
    -float *params: Serie de datos estilizados.
    -float *forecast: Prediccion Calculada.
*/
float forecast_arima(float *series, int n, ARIMA_Model *model, float *params) {
    float forecast = 0.0f;
    // Componente AR
    #pragma omp parallel for reduction(+:forecast)
    for (int i = 0; i < model->p; i++) {
        forecast += params[i] * series[n - 1 - i];
    }
    // Componente MA
    forecast += params[model->p] * (series[n-1] - mean(series, n));
    return forecast;
}
```

Función forecast_arima

Esta función es la que calcula media móvil y el componente autorregresivo el valor obtenido de esta función es nuestra predicción para el siguiente valor en la serie pasada desde el JSON, es decir es nuestra predicción de que sucederá a razón de lo acontecido hasta el momento.

JSONParser

Ciertamente, aquí la parte a analizar principalmente es el main porque es lo que diferencia a ambos archivos principalmente.

Saltándonos la parte de inicialización y apertura del fichero JSON, ya que lo único que tiene de especial es que dejamos esa tarea al proceso de rango 0 al igual que él parseó inicial del fichero.

Comenzamos ya con el uso de MPI a partir de un primer broadcast de lo que es la cantidad de campos que tenemos que buscar; los cuales en caso de error a la hora de enviarse podremos abortar la ejecución.

```c
//Envio del numero de campos a los demas procesos
MPI_Bcast(&num_campos, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Envío del tamaño del struct

Dicho esto, lo siguiente en ser enviado serán en sí qué valor tienen los campos que estamos buscando en el JSON

```c
//Enviamos los campos a buscar
MPI_Bcast(campos, num_campos * 32, MPI_CHAR, 0, MPI_COMM_WORLD);
```

Envió de las etiquetas de los campos

Este envío lo haremos con un buffer de 32 bytes por la cantidad de datos que estamos esperando estamos suponiendo que ninguna etiqueta tendrá más de 32 caracteres.

```
//Reparto de las lineas entre los distintos procesos.
int registros_por_rank = total_registros / size;
int resto = total_registros % size;
int inicio = rank * registros_por_rank + (rank < resto ? rank : resto);
int fin = inicio + registros_por_rank + (rank < resto ? 1 : 0);
int registros_locales = fin - inicio;

//Asignacion de hilos OMP
omp_set_num_threads(nh);
```

Reparto del trabajo.

Una vez hemos indicado esos dos primeros valores al igual que la cantidad de registros que hay en total a cada hilo se procede en esas líneas al reparto de la tarea siendo primero definir a cuantos bloques de datos tocamos por proceso, y una guarda para evitar que ningún segmento quede fuera del procesamiento; terminamos diciendo que el inicio será rango del proceso MPI * cantidad a la que tocamos + en caso de ser el rango menor que el resto es decir si el resto no es 0 se le asigna parte de lo que sería el resto. Y algo muy parecido pasará para marcar el fin del reparto.

```
        }

        #pragma omp parallel for
        for (int j = 0; j < registros_locales; j++) {
            cJSON *item = cJSON_GetArrayItem(root, inicio + j);
            if (item) {
                cJSON *valor = cJSON_GetObjectItem(item, campos[i]);
                datos_locales[i].valores[j] = convertir_valor(valor);
            } else {
                datos_locales[i].valores[j] = 0.0f;
            }
        }
    }
```

Análisis del segmento propio.

A cada fichero se le pasa una parte del JSON en la que iterara; además como os habréis percatado en la captura se puede observar que paralelizamos ese bucle de captura para cada proceso acelerando aún más el mecanismo de captura de datos.

```
for (int i = 0; i < num_campos; i++) {
    MPI_Gatherv(
        datos_locales[i].valores,
        registros_locales,
        MPI_FLOAT,
        rank == 0 ? valores_completos[i] : NULL,
        recv_counts,
        displs,
        MPI_FLOAT,
        0,
        MPI_COMM_WORLD
    );
```

Recolección.

Tras dos secciones de inicialización para lo que serían las estructuras locales de cada proceso con sus respectivos posibles frenos en caso de error; vemos las funciones Gatherv que hará en la comunicación de los procesos ahora listos para el cálculo final del cual se encargará únicamente el proceso 0 llamando a las funciones antes explicadas en Arima.

```c
// Procesamiento en el proceso 0
if (rank == 0) {
    float *dias_transcurridos = array_auxiliar_dias(total_registros);

    // Regresión Lineal para cada campo
    for (int i = 0; i < num_campos; i++) {
        float b0, b1;
        linear_regression(dias_transcurridos, valores_completos[i], total_registros, &b1, &b0);
        float prediccion = b0 + b1 * (total_registros + 1);
        printf("Regresión Lineal - Previsión %s día %d: %.2f\n",
               campos[i], total_registros + 1, prediccion);
    }

    // Modelos ARIMA
    ARIMA_Model modelos_arima[] = {
        {1, 0, 0},  // AR(1)
    };

    int num_modelos = sizeof(modelos_arima) / sizeof(modelos_arima[0]);

    for (int m = 0; m < num_modelos; m++) {
        ARIMA_Model modelo = modelos_arima[m];
        printf("\nAplicando modelo ARIMA(%d,%d,%d) a todos los campos:\n",
               modelo.p, modelo.d, modelo.q);

        for (int i = 0; i < num_campos; i++) {
            float *params = malloc((modelo.p + modelo.q) * sizeof(float));
            if (!params) {
                fprintf(stderr, "Error al asignar memoria para parámetros ARIMA\n");
                continue;
            }

            fit_arima(valores_completos[i], total_registros, &modelo, params);
            float prediccion = forecast_arima(valores_completos[i], total_registros, &modelo, params);
            printf("  Campo %s - Previsión día %d: %.2f\n",
                   campos[i], total_registros + 1, prediccion);

            free(params);
        }
    }
```

Cálculo de las predicciones.

En este último fragmento del código observaréis las llamadas a las funciones del fichero Arima; usaremos la regresión lineal como una línea base para pensar en cómo sería la predicción, pero no debemos olvidar la naturaleza de ese cálculo, puesto que este analiza la tendencia de la serie es por ello por lo que siempre tenderá a valores en el alza o la muy baja y en caso de tener un gran volumen de datos su predicción será inútil.

Hibrido OMP CUDA

De cara a hablar acerca de esta versión del código donde usamos el poder de los Kerneles de CUDA, he sentido la necesidad incrementar un poco la intensidad aritmética en algunas funciones de Arima haciendo algo más compleja la implementación de ciertos fragmentos que desarrollaré más tarde.

Arima

En esta versión la parte de Arima he buscado un poco más de intensidad aritmética en los procesos donde se pudiese lograr tratando las funciones en orden vemos lo siguiente:

Para analizar el kernel de la regresión lineal lo separaremos en 3 fases; inicialización, cálculo y rectificación.

```
__global__ void linear_regression_kernel(float *x, float *y, int n, float *b1, float *b0)
{
    extern __shared__ float sdata[];
    float *sx = sdata;
    float *sy = &sdata[blockDim.x];

    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;

    // Carga de datos en memoria compartida
    if (i < n)
    {
        sx[tid] = x[i];
        sy[tid] = y[i];
    }
    else
    {
        sx[tid] = 0.0f;
        sy[tid] = 0.0f;
    }
    __syncthreads();

    // Cálculo de medias
    float sum_x = 0.0f, sum_y = 0.0f;
    for (int s = 0; s < blockDim.x; s++)
    {
        if (blockIdx.x * blockDim.x + s < n)
        {
            sum_x += sx[s];
            sum_y += sy[s];
        }
    }

    float mean_x = sum_x / n;
    float mean_y = sum_y / n;
```

Kernel regresión lineal inicialización

En esta primera fase cargamos a la memoria compartida lo valores que hemos obtenido antes en él parseo del JSON o lo sacado por la función “array_auxiliar_dias”; además del primer cálculo de la media de los valores de ambos arrays. Llamando al kernel anterior, estos haciendo la suma en la memoria compartida, nos aseguramos que se haga la media de todos los valores, no solo los pertenecientes al bloque.

```
// Cálculo de numerador y denominador
float num = 0.0f, den = 0.0f;
if (i < n)
{
    float x_diff = x[i] - mean_x;
    num = x_diff * (y[i] - mean_y);
    den = x_diff * x_diff;
}

// Almacenar resultados en memoria compartida para reducción
sx[tid] = num;
sy[tid] = den;
__syncthreads();

// Reducción para sumar todos los hilos
for (int s = blockDim.x / 2; s > 0; s >>= 1)
{
    if (tid < s)
    {
        sx[tid] += sx[tid + s]; // Suma numerador
        sy[tid] += sy[tid + s]; // Suma denominador
    }
    __syncthreads();
}
```

Kernel Regresión lineal cálculo.

Seguimos con el cálculo de los numeradores y denominadores que se guardaran de nuevo en la memoria compartida que tras un punto de sincronización haremos una reducción horizontal de todos los valores para tener el valor real y efectivo para el total completo.

```
    if (tid == 0)
    {
        // Evitar división por cero
        if (sy[0] != 0.0f)
        {
            b1[blockIdx.x] = sx[0] / sy[0];
        }
        else
        {
            b1[blockIdx.x] = 0.0f;
        }
        b0[blockIdx.x] = mean_y - (b1[blockIdx.x] * mean_x);

        // Opcional: añadir limitación a las predicciones para evitar valores extremos
        if (isnan(b0[blockIdx.x]) || isinf(b0[blockIdx.x]))
            b0[blockIdx.x] = 0.0f;
        if (isnan(b1[blockIdx.x]) || isinf(b1[blockIdx.x]))
            b1[blockIdx.x] = 0.0f;
    }
}
```

Kernel regresión lineal rectificación.

Como comentamos antes en este documento la regresión lineal básica nos puede servir como orientación de cómo se debería comportar, pero debido a la naturaleza con una gran cantidad de valores termina tendiendo a volverse difusa o incluso altamente incorrecta.

```cpp
/*
Descripcion:
    -Funcion para la diferenciacion o alisado de los datos en busqueda de generar series estacionarias
Parametros:
    -float *series: Array con los valores a normalizar
    -int n: Tamaño del array original
    -int d: Factor de diferenciación.
    -float *diff_series: Array contenedor de la nueva serie estilizada.
*/
__global__ void difference_kernel(float *series, int n, int d, float *diff_series)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= d && i < n)
    {
        diff_series[i - d] = series[i] - series[i - d];
    }
}
```

Kernel Diferenciación

Este es tal vez el procedimiento que entre la versión OMP y la CUDA menos cambie entre otras cosas porque este cambia el pragma OMP por conocer su posición como Threads en el conjunto global

```cpp
/*
Descripcion:
    -Funcion para el ajuste de los datos, tras la diferenciacion de los mismo.
Parametros:
    -float *series: Array de datos previamente diferenciados.
    -int n: Tamaño de la serie tras la diferenciación.
    -ARIMA_Model *model: Modelo ARIMA que estamos usando
    -float *params: Array contenedor de la serie ajustada
*/
__global__ void fit_arima_kernel(float *series, int n, ARIMA_Model *model, float *params)
{
    // Better initialization using autocorrelation
    if (threadIdx.x < model->p)
    { // AR terms
        if (blockIdx.x * blockDim.x + threadIdx.x < n - 1)
        {
            params[threadIdx.x] = 0.8f * (threadIdx.x + 1) / model->p; // autocorrelaion
        }
    }
    else if (threadIdx.x < model->p + model->q)
    { // MA terms
        params[threadIdx.x] = 0.2f * (threadIdx.x - model->p + 1) / model->q;
    }
}
```

Kernel fit_arima

Este kernel si ve cambios porque tenemos más capacidad de cómputo para el alisado de los datos usamos términos de autocorrelación y la media móvil para este primer ajuste aplicándolos de manera ponderada.

```
*/
__global__ void forecast_arima_kernel(float *series, int n, ARIMA_Model *model,
                                      float *params,float *forecast)
{
    float ar_component = 0.0f;
    float ma_component = 0.0f;

    // AR Component
    for (int i = 0; i < model->p; i++)
    {
        ar_component += params[i] * series[n - 1 - i];
    }

    // MA Component
    float residual = series[n - 1] - ar_component;
    ma_component = params[model->p] * residual;

    if (threadIdx.x == 0)
    {
        forecast[0] = ar_component + ma_component;
        // Gestion de valores extremos. (Opcional)
        forecast[0] = fmaxf(0.0f, fminf(100.0f, forecast[0]));
    }
}
```

Kernel ARIMA

Este realiza el mismo cálculo que realizaba su versión OMP solo que tiene en cuenta un residuo de las series de diferenciación con respecto al parámetro autorregresivo.

JSONParser

```c
CampoDatos *extraer_campos_openmp(cJSON *array_json, char **campos, int num_campos, int *total_registros)
{
    *total_registros = cJSON_GetArraySize(array_json);
    CampoDatos *resultados = (CampoDatos *)malloc(num_campos * sizeof(CampoDatos));

#pragma omp parallel for
    for (int i = 0; i < num_campos; i++)
    {
        resultados[i].nombre = strdup(campos[i]);
        resultados[i].valores = (float *)malloc(*total_registros * sizeof(float));
        resultados[i].cantidad = *total_registros;

#pragma omp parallel for
        for (int j = 0; j < *total_registros; j++)
        {
            //Obtiene el conjunto completo contenido en {}
            cJSON *item = cJSON_GetArrayItem(array_json, j);
            //obtiene la correlacion clave valor con respecto al campo buscado
            cJSON *valor = cJSON_GetObjectItem(item, campos[i]);

            if (valor)
            {
                if (cJSON_IsNumber(valor))
                {
                    resultados[i].valores[j] = (float)valor->valuedouble;
                }
                else if (cJSON_IsString(valor))
                {
                    char str_valor[32];
                    strcpy(str_valor, valor->valuestring);
                    cambiar_comas_por_puntos(str_valor);
                    resultados[i].valores[j] = atof(str_valor);
                }
                else
                {
                    resultados[i].valores[j] = 0.0f;
                }
            }
            else
            {
                resultados[i].valores[j] = 0.0f;
            }
        }
    }
```

Función extracción de datos

Aquí vemos la primera diferencia con respecto a la versión MPI, aunque se realizaba la misma función debido a la naturaleza de MPI esta estaba segmentada por el código del main, aquí, sin embargo, compone una funcionalidad extraíble sin tener que tocar el programa principal.

Pero esta no será la última sorpresa que se encuentra dentro de este fichero.

```
/*
Descripción:
    - Funcion para el lanzamiento de  los kernels CUDA, ocupada de la reserva y asignacion de los recursos para que la GPU pueda ocuparse del trabajo.
Parametros:
    -float *d_dias: Array de enteros auxiliar para el cálculo de la regresión lineal, componente independiente del calculo.
    -float *d_registros: Valores tomados del JSON.
    -int n: Numero total de registros.
    -float *h_b0: Espacio en el host donde copiar el valor calculado en el device
    -float *h_b1: Espacio en el host donde copiar el valor calculado en el device

*/
void perform_linear_regression(float *d_dias, float *d_registros, int n, float *h_b0, float *h_b1)
{
    float *d_b0, *d_b1;

    // Allocate device memory for results
    checkCudaErrors(cudaMalloc((void **)&d_b0, sizeof(float)));
    checkCudaErrors(cudaMalloc((void **)&d_b1, sizeof(float)));

    // Configure kernel launch parameters
    int threads = 256;
    int blocks = (n + threads - 1) / threads;
    size_t shared_mem = 2 * threads * sizeof(float);

    // Launch kernel
    linear_regression_kernel<<<blocks, threads, shared_mem>>>(d_dias, d_registros, n, d_b1, d_b0);

    // Copy results back to host
    checkCudaErrors(cudaMemcpy(h_b0, d_b0, sizeof(float), cudaMemcpyDeviceToHost));
    checkCudaErrors(cudaMemcpy(h_b1, d_b1, sizeof(float), cudaMemcpyDeviceToHost));

    // Free device memory
    cudaFree(d_b0);
    cudaFree(d_b1);
}
```

Lanzamiento Kernel regresión lineal

Por legibilidad para el main esta parte del proceso también se separó del flujo de lectura principal; separamos por aquí la reserva de los campos de memoria que necesitaremos para lanzar el kernel además de inicializar el grid y el número de hilos por bloque también aislamos aquí la liberación y copia de los valores obtenidos en la GPU para los cálculos requeridos.

```
/*
Descripción:
    - Funcion para el lanzamiento de  los kernels CUDA, ocupada de la reserva y asignacion de los recursos para que la GPU pueda ocuparse del trabajo.
Parametros:
    -float *d_series: Array de enteros que será la serie estilizada por la funcion difference
    -int n: Numero total de registros.
    -ARIMA_Model *model: Modelo de ARIMA que usaremos para la prediccion.
    -float *h_forecast : Espacio en el host donde copiar el valor calculado en el device
*/
void perform_arima_forecast(float *d_series, int n, ARIMA_Model *model, float *h_forecast)
{
    float *d_forecast, *d_params;
    ARIMA_Model *d_model;

    // Allocate device memory
    checkCudaErrors(cudaMalloc((void **)&d_forecast, sizeof(float)));
    checkCudaErrors(cudaMalloc((void **)&d_params, (model->p + model->q) * sizeof(float)));
    checkCudaErrors(cudaMalloc((void **)&d_model, sizeof(ARIMA_Model)));

    // Copy model to device
    checkCudaErrors(cudaMemcpy(d_model, model, sizeof(ARIMA_Model), cudaMemcpyHostToDevice));

    // Configure kernel launch parameters
    int threads = 256;
    int blocks = 1;
    size_t shared_mem = n * sizeof(float);

    // Fit ARIMA model
    fit_arima_kernel<<<(model->p + model->q + threads - 1) / threads, threads>>>(d_series, n, d_model, d_params);

    // Forecast
    forecast_arima_kernel<<<blocks, threads, shared_mem>>>(d_series, n, d_model, d_params, d_forecast);

    // Copy result back to host
    checkCudaErrors(cudaMemcpy(h_forecast, d_forecast, sizeof(float), cudaMemcpyDeviceToHost));

    // Free device memory
    cudaFree(d_forecast);
    cudaFree(d_params);
    cudaFree(d_model);
}
```

Lanzamiento kernel Arima

Al igual que antes para ayudarnos con la legibilidad del código consideramos el usar una función que aislé del resto del código el lanzamiento de nuestro kernel y el respectivo

cálculo de todas las fases que son necesarias para nuestro uso de ARIMA como modelo de predicción.

Componentes adicionales

Este apartado nace de la necesidad de crear unos datos sintéticos para el programa, ya que usar CUDA en un ejemplo tan pequeño como el que estábamos usando como modelo base para OMP y MPI; es un poco desperdiciar la potencia del sistema porque pasará más tiempo haciendo copias de valores de una memoria a la otra antes que realizar verdaderamente haciendo uso de la potencia de cálculo.

Extensor de Datos

```python
def main():
    """
    Función principal para ejecutar el script desde la línea de comandos.
    """
    parser = argparse.ArgumentParser(description='Extender datos meteorológicos hasta una fecha específica.')
    parser.add_argument('input_file', help='Archivo JSON de entrada con los datos meteorológicos')
    parser.add_argument('output_file', help='Archivo JSON de salida con los datos extendidos')
    parser.add_argument('fecha_final', help='Fecha final para la extensión (formato YYYY-MM-DD)')

    args = parser.parse_args()

    # Leer el archivo de entrada con codificación ISO-8859-1 para manejar caracteres españoles
    with open(args.input_file, 'r', encoding='iso-8859-1') as f:
        datos = json.load(f)

    # Generar datos extendidos
    datos_extendidos = generar_datos_extendidos(datos, args.fecha_final)

    # Escribir el archivo de salida
    with open(args.output_file, 'w', encoding='utf-8') as f:
        json.dump(datos_extendidos, f, indent=2, ensure_ascii=False)

    print(f"Datos extendidos hasta {args.fecha_final} y guardados en {args.output_file}")
```

Main extensor datos

Para la creación de datos sintéticos empezamos por analizar el main donde haremos la toma de argumentos que necesitaremos para el uso del script seguido por la apertura del JSON con codificación para tolerar las “Ñ” en caso de que sean necesarias como es el caso con nuestros datos de ejemplo porque son de La Coruña.

Continúa por llamar a la función extensora y por último hará la carga tanto de lo leído en el fichero original como lo generado por el script en un fichero con el nombre que le hayamos dado

```
    list: Lista de diccionarios con los datos originales y los datos extendidos.
    """
    # Convertir fechas a objetos datetime
    fecha_ultima = datetime.strptime(datos_originales[-1]['fecha'], "%Y-%m-%d")
    fecha_final = datetime.strptime(fecha_final_str, "%Y-%m-%d")

    # Verificar que la fecha final sea posterior a la última fecha en los datos
    if fecha_final <= fecha_ultima:
        print("La fecha final debe ser posterior a la última fecha en los datos.")
        return datos_originales

    # Preparar datos para análisis de tendencias
    fechas = [datetime.strptime(d['fecha'], "%Y-%m-%d") for d in datos_originales]
    dias_del_ano = [d.timetuple().tm_yday for d in fechas]

    # Extraer valores históricos por día del año
    datos_por_dia = {}
    for i, dia in enumerate(dias_del_ano):
        if dia not in datos_por_dia:
            datos_por_dia[dia] = []
        datos_por_dia[dia].append(datos_originales[i])

    # Calcular estadísticas por día del año
    estadisticas_por_dia = {}
    for dia in datos_por_dia:
        datos_dia = datos_por_dia[dia]
```

Extensor datos gestor fechas

Comenzaremos nuestra extensión para hacerla de manera correcta con una gestión de las fechas para asegurarnos que el límite dado sea posterior a lo ya tenido en el JSON.

Ahí podremos percibir el inicio a su vez de la extracción de la tendencia y análisis de valores históricos, ya que este script no genera los datos de manera aleatoria, sino que se basa en lo que observa dentro del JSON anterior.

```python
        prob_lluvia = sum(1 for p in prec_values if p > 0) / len(prec_values)

        # Para temperaturas y velocidad del viento
        tmed_values = [float(d['tmed'].replace(',', '.')) for d in datos_dia]
        tmax_values = [float(d['tmax'].replace(',', '.')) for d in datos_dia]
        tmin_values = [float(d['tmin'].replace(',', '.')) for d in datos_dia]
        velmedia_values = [float(d['velmedia'].replace(',', '.')) for d in datos_dia]

        # Guardar estadísticas calculadas
        estadisticas_por_dia[dia] = {
            'prob_lluvia': prob_lluvia,
            'prec_mean': np.mean(prec_values),
            'prec_std': np.std(prec_values),
            'tmed_mean': np.mean(tmed_values),
            'tmed_std': np.std(tmed_values),
            'tmax_mean': np.mean(tmax_values),
            'tmax_std': np.std(tmax_values),
            'tmin_mean': np.mean(tmin_values),
            'tmin_std': np.std(tmin_values),
            'velmedia_mean': np.mean(velmedia_values),
            'velmedia_std': np.std(velmedia_values)
        }
```

Cálculo de media y desviación estándar

Claro para mantenernos en un margen día a día calcularemos con la ayuda de la librería NumPy la desviación estándar y la media para todo aquel valor que sean calculables. Consta decir que antes de este cálculo también haremos la ya clásica transformación en este proyecto de pasar los caracteres “,” a “.”.

```python
# Generar precipitación (distribución mixta)
if random.random() < stats_dia['prob_lluvia']:
    prec = max(0, np.random.normal(stats_dia['prec_mean'], stats_dia['prec_std']))
    prec = round(max(0, prec), 1)  # Redondear a 1 decimal y asegurar que no sea negativo
    if prec < 0.1 and random.random() < 0.1:  # 10% de probabilidad de "Ip"
        prec = "Ip"
else:
    prec = 0.0

# Generar temperaturas y velocidad del viento con distribución normal
tmed = round(np.random.normal(stats_dia['tmed_mean'], stats_dia['tmed_std']), 1)
tmax = round(np.random.normal(stats_dia['tmax_mean'], stats_dia['tmax_std']), 1)
tmin = round(np.random.normal(stats_dia['tmin_mean'], stats_dia['tmin_std']), 1)
velmedia = round(max(0, np.random.normal(stats_dia['velmedia_mean'], stats_dia['velmedia_std'])), 1)

# Asegurar que tmax >= tmed >= tmin
tmed = min(tmax, max(tmin, tmed))
```

Generación del nuevo valor

Dado que las precipitaciones se comportan de manera algo más irregular su cálculo lo haremos con una distribución mixta, en cambio, el resto de los valores si podemos usar la distribución normal además de añadir una guarda para asegurarnos que las temperaturas nuevas calculadas sean congruentes entre sí. Por último, no deja de ser un print estilizado para que se carguen esos valores en el JSON.

Makefile y Readme

Todos los scripts cuentan con un make file para facilitar tanto la compilación como la ejecución de estos, tienen una serie de parámetros predeterminados que pueden ser cambiados con el comando export. En caso de duda dentro de cada versión y el extensor hay un Readme que explica con más detenimiento el funcionamiento del make y los parámetros mutables.

Output

Analizando la salida del programa, aunque esta pueda ser relativamente sencilla de entender hay ciertos factores que son importantes detallar como son los siguiente.

Campos extraíbles

Primero me parece que es interesante detallar que significan cada uno de los campos mas importantes que podemos obtener con nuestro programa siendo en una lista los siguientes:

- ***tmax*** Temperatura máxima.

- ***tmed*** Temperatura media.

- ***tmin*** Temperatura mínima.

- ***velmedia*** Velocidad del viento media.

- ***prec*** Precipitaciones.

- ***sol*** Incidencia ultravioleta.

- ***presmax*** Presión atmosférica máxima.

- ***presmin*** Presión atmosférica máxima.

Salida CUDA

```
----------------------------------------
Input: ./datosEx.json
Campos a analizar: tmax tmed tmin prec velmedia
Numero de Hilos por proceso: 4
----------------------------------------
./PredictorCuda ./datosEx.json 4 tmax tmed tmin prec velmedia
Regresión Lineal - Previsión tmax día 3973: 398.05
Regresión Lineal - Previsión tmed día 3973: 331.71
Regresión Lineal - Previsión tmin día 3973: 265.42
Regresión Lineal - Previsión prec día 3973: 92.73
Regresión Lineal - Previsión velmedia día 3973: 81.93
Modelo ARIMA(1,0,0)
Previsión tmax día 3973: 14.72
Previsión tmed día 3973: 12.32
Previsión tmin día 3973: 9.92
Previsión prec día 3973: 3.76
Previsión velmedia día 3973: 2.80
```

Salida Versión CUDA

Aunque las diferencias entre las versiones es mínima ya que realmente están estilizadas de la misma manera. Es importante destacar la primera cabecera que nos dice en este caso que campos son los que estamos buscando dentro de nuestro fichero, de que fichero estamos obteniendo dichos valores y como en esta versión contamos con OpenMP también cuantos hilos se están generando en las zonas paralelas de OMP.

Viendo la salida en si nos damos cuenta de que existen 2 secciones diferenciadas por la cabecera que precede a la previsión, que tienen el siguiente formato

$$Regresion\ Lineal$$
$$-\ prevision\ "Campo_{concreto}"\ día\ "proximovalor_{secuencia}":"Predicción"$$

Mientras que para la parte de ARIMA veremos que se tiene lo siguiente:

$$Modelo(p,d,q)Previsión\ "Campo_{concreto}\ "día\ "proximovalor_{secuencia}":"Predicción"$$

Esta estructura para la parte de predicciones se repetirá para la versión MPI OpenMp de la cual hay que destacar lo siguiente:

```
----------------------------------------
Ejecutado en 4 procesos
Input: ./datos.json
Campos a analizar: tmax tmed tmin prec velmedia
Numero de Hilos por proceso: 4
----------------------------------------
mpirun -np 4 ./PredictorHibrido ./datos.json 4 tmax tmed tmin prec velmedia
Regresión Lineal - Previsión tmax día 143: 13.71
Regresión Lineal - Previsión tmed día 143: 10.97
Regresión Lineal - Previsión tmin día 143: 8.24
Regresión Lineal - Previsión prec día 143: 6.48
Regresión Lineal - Previsión velmedia día 143: 3.84

Aplicando modelo ARIMA(1,0,0) a todos los campos:
  Campo tmax - Previsión día 143: 6.70
  Campo tmed - Previsión día 143: 5.40
  Campo tmin - Previsión día 143: 4.15
  Campo prec - Previsión día 143: 6.00
  Campo velmedia - Previsión día 143: 3.45
```

Salida Versión MPI OMP

Ahora en la cabecera aparece una nueva línea que nos dice cuantos procesos MPI se encuentran trabajando en nuestro programa.

```
adrian@Ercamarero:/mnt/c/Users/adria/Downloads/DatosAemet/Extensor_Datos$ make run
Ejecutando el programa...
Datos extendidos hasta 2026-01-28 y guardados en ./output.json
```

Salida ExtensorDatos.py

La salida de este es bastante mas sencilla ya que por la línea de comandos solo nos devuelve hasta que fecha hemos extendido los datos y como se llama el nuevo fichero por ello y como ya se ha mencionado muchas veces pero no se ha mostrado nunca el JSON del que extraemos los datos ponemos una muestra generada por nuestro extensor de datos:

Estructura de Nuestro JSON