Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática

Autor: Esteban Aspe Ruiz
Tutor: José Antonio Mateo Cortés

Madrid, 3 de junio de 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Implementación y Evaluación de un Algoritmo de Altas Prestaciones para el Cálculo de la Anomalía Climática

3 de junio de 2024

Autor: Esteban Aspe Ruiz
Tutor: José Antonio Mateo Cortés
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

El siguiente trabajo consiste en el desarrollo de un programa de alto rendimiento que permita el cálculo de la anomalía climática para grandes superficies. La anomalía climática es un factor esencial para estimar la intensidad de un posible incendio, y en su cálculo son necesarios al menos 30 años de datos históricos diarios. Esto provoca que sin un programa de alto rendimiento, su cálculo en minutos sería imposible.

Para lograrlo, se ha diseñado a su vez un programa de análisis, filtrado y transformación del conjunto de datos, creando así un conjunto de datos optimizado para el cálculo de la anomalía climática, asegurando que el cálculo sea lo más preciso posible. Además, en el programa de cálculo, se han desarrollado distintas implementaciones utilizando distintas tecnologías que permiten aprovechar el paralelismo de los procesadores y la potencia de las tarjetas gráficas, buscando el máximo rendimiento posible en este programa.

Abstract

The following work consist of the development of a high performance software that allows the calculation of the climate anomaly for large areas. Climate anomaly is an essential factor for the estimation of the intensity of a possible fire, and, for its calculation, 30 years of historical daily data is needed. As a consequence, its calculation in a few minutes would be impossible without a high performance software.

To achieve this, complementary software has been developed to analyze, filter, and transform the dataset to create a new optimized dataset to calculate the climate anomaly and to guarantee a precise calculation. Moreover, in the calculation software, different implementations have been developed using different technologies that allow to take advantage of the parallelism of processors and to exploit the power of graphics cards, seeking to max the performance of this software.

Tabla de contenidos

Capítulo 1

Introducción

1.1. Motivación y justificación

Debido al cambio climático y su consecuente aumento en la temperatura, las sequías se han hecho más frecuentes y severas, aumentando a su vez la duración, severidad y frecuencia de los incendios forestales[1]. Como respuesta a este hecho, el técnico analista de incendios Fernando Chico Zamora presentó en el 8º Congreso Forestal Español en 2022 un índice de predicción del peligro de incendios forestales. Este índice se basa en las anomalías que se producen en la temperatura media y en la sequía, y combinadas conforman la Anomalía Climática. Al tratarse de un anomalía, es necesario partir de un punto pasado en el tiempo. Estas anomalías se calculan a partir de percentiles, y Fernando Chico determina que, para que los percentiles de temperatura media y sequía sean los suficientemente representativos, son necesarios datos históricos diarios de al menos 30 años. En consecuencia, el volumen de datos mínimo para este cálculo resulta muy grande, provocando que en un programa convencional se podría llegar a tardar semanas o incluso meses en hacer estos cálculos en un superficie grande y con una precisión suficiente. Por ello, es necesario desarrollar un programa capaz de calcular la anomalía climática en un tiempo razonable.

1.2. Objetivo

El objetivo de este trabajo es implementar un programa que consiga calcular la anomalía climática en un tiempo razonable de una gran superficie y con una precisión suficiente. Para maximizar el rendimiento, el programa deberá aprovechar el paralelismo dentro del procesador y la potencia de las tarjetas gráficas (en inglés GPU). Además, debido al gran volumen de datos que se generará, el programa deberá crear un archivo con la capacidad de comprimirse a un tamaño razonable.

1.3. Estructura de la memoria

La memoria contará de 5 capítulos además de la introducción. A continuación, se resumirá brevemente el contenido de todos ellos.

2. Estado del Arte: En este capítulo se expondrán los conceptos básicos para la comprensión del proyecto, los proyectos similares que se han encontrado y las tecnologías que se usan en la actualidad y son utilizadas en el proyecto.
3. Desarrollo: Se detallará todo el desarrollo del proyecto, incluyendo las dificultades y errores que se encontraron, así como el funcionamiento de las distintas implementaciones desarrolladas.
4. Evaluación: Se analizarán los tiempos obtenidos por las distintas implementaciones y las posibles limitaciones de las mismas.
5. Conclusiones y trabajo futuro: Se analizará brevemente si se han conseguido los objetivos establecidos y los resultados obtenidos.
6. Análisis de Impacto: Se analizarán las consecuencias y aportes de este trabajo a la problemática anteriormente descrita, y su implicación con los objetivos de sostenibilidad de la Organización de Naciones Unidas (ONU).

Capítulo 2

Estado del Arte

En este capítulo se explicarán los conceptos básicos en los que se basa el trabajo, la mención de trabajos relacionados y algunas de la tecnologías que son utilizadas en la actualidad para la Computación de Alto Rendimiento (en inglés, HPC) y las que han sido utilizadas en el desarrollo de este proyecto.

2.1. ¿Qué es la anomalía climática?

Tal y como se mencionó en la motivación, este concepto nace a partir del artículo publicado por el técnico analista de incendios Fernando Chico Zamora en el 8º Congreso Forestal Español en 2022, denominado 'El índice de predicción del peligro de incendios forestales utilizado por el INFOCAM', el acrónimo del plan especial de emergencias por incendios forestales de Castilla-La Mancha[2]. En dicho documento define el Índice de Propagación Potencial (IPP), un indicador numérico de máximos, que caracteriza la intensidad, el desarrollo y la dificultad de extinción que un incendio forestal podría llegar a alcanzar en un momento y lugar determinados. Este índice esta compuesto por una parte meteorológica y una parte climática, que es en la que se enfocará este trabajo. La parte meteorológica se obtiene a partir de los datos proporcionados por los pronósticos procedentes de los modelos numéricos de predicción meteorológica. A este factor del índice lo denomina Índice de Peligro Meteorológico (IMP). La parte climática es sensible al nivel de anomalía climática existente y está basada en la observación meteorológica. A este factor lo denomina Anomalía Climática. Multiplicando ambas, se calcula el Índice de Propagación Potencial.

La anomalía climática, que será lo que se calculará en este trabajo, se basa en el hecho de que los episodios de incendios forestales más graves se producen cuando se dan dos situaciones climáticas anómalas. Estas situaciones son cuando los niveles de sequía son más altos que los valores medios históricos y cuando la temperatura media en los días previos está por encima de los valores medios históricos. Tal y como se menciona en el artículo, los peores episodios de incendios en Castilla-La Mancha coinciden con años en los que los niveles de sequía eran muy altos y se producen simultáneamente episodios con temperatura muy por encima de la media histórica. Estas dos situaciones están

estrechamente relacionadas con los conceptos de estrés hídrico, que representa la sequía, y el estrés térmico, que representa las temperaturas muy elevadas. Ambas representan que cuando tienen valores muy altos, la actividad fisiológica de ciertas especies vegetales se ven alteradas, aumentando su capacidad de participar como combustible en un incendio.

Para calcularlos, Fernando Chico establece que el estrés térmico se calcula con el percentil de la temperatura media de los últimos 8 días para cada día determinado del año. El periodo de referencia a tomar para el cálculo del estrés térmico intenta representar la inercia respecto a la variación de la disponibilidad que tienen los combustibles forestales. El estrés hídrico se estima a partir de percentiles del *drought code*, que forma parte del índice canadiense [3]. Finalmente, la anomalía climática es el valor medio entre el estrés hídrico y el estrés térmico.

2.2. Proyectos similares

Aunque no se ha encontrado ningún proyecto que realice un cálculo completo de la anomalía climática o el Índice de Propagación de un Incendio, se han encontrado otros proyectos que realizan parte de los cálculos necesarios.

2.2.1. Aplicación web que permite el cálculo de la evapotranspiración en una determinada zona

Este proyecto es el trabajo de fin de grado (TFG) de Javier Alarcón, alumno de la escuela de Ingeniería Informática de la Universidad de Castilla-La Mancha [4]. En él, se desarrolla una aplicación web que permite seleccionar unas determinadas coordenadas, y devuelve el cálculo de la evapotranspiración en ese momento a través de los datos meteorológicos obtenidos con llamadas a la API de la Agencia Estatal de Meteorología (AEMET). Para el cálculo de la anomalía climática, en concreto para el estrés hídrico, es necesario el cálculo de la evapotranspiración para calcular el *drought code*. Sin embargo, para el *drought code* es necesario calcular la evapotranspiración potencial, mientras que en este trabajo de fin de grado se utiliza el cálculo de la evapotranspiración de referencia. Tal y como se describe en este documento utilizado en la E.T.S de Ingenieros Agrónomos de la UPM [5], a pesar de tener distintas definiciones, la principal diferencia ocurre en distintos cultivos que con zonas áridas y semiáridas pueden resultar en distintos resultados, por lo que se decidió definir la evapotranspiración de referencia para cultivos. Otra de las principales diferencias de este trabajo de fin de grado es que está enfocado a calcular en el momento de la petición la evapotranspiración, mientras que para la anomalía climática es necesario calcularlo masivamente con datos históricos. Debido a esto, el TFG de Alarcón no esta orientado a la Computación de Alto Rendimiento (en inglés HPC), a diferencia de este TFG.

2.3. Fuentes de Datos

Como se comentó en el Apartado 2.1, para calcular la anomalía climática se necesitan al menos 30 años de datos históricos diarios de precipitación y temperatura media. Se encontraron dos fuentes fiables que proporcionaban esta cantidad de datos, la Agencia Estatal de Meteorología (AEMET) y Copernicus, un programa de la Unión Europea.

2.3.1. Datos en AEMET

La AEMET [6] proporciona, además de datos actualizados en tiempo real, datos orientados a la predicción del clima futuro [7]. Estos datos, procedentes del Banco Nacional de Datos climatológicos y de las estaciones de la AEMET, contienen datos interpolados diarios sobre la precipitación y la temperatura máxima y mínima con una precisión de máxima de 5 km^2 desde 1950 hasta 2022. Estos datos se proporcionaban en formato texto con un fichero maestro con las coordenadas de los puntos y en formato netCDF. A pesar de ser una gran cantidad de datos, se consideró insuficiente para el ejemplo más grande. Esto, sumado a la inconveniencia de no otorgar la temperatura media diaria, sino la temperatura máxima y mínima, llevó a la decisión de descartar el uso de estos datos.

2.3.2. Datos en Copernicus

Copernicus [8] es el Programa de Observación de la Tierra de la Unión Europea, y ofrece una gran cantidad de datos tantos procedentes de satélites como *in situ* de todo Europa. Copernicus, de forma similar a la AEMET, proporciona datos *in situ* diarios de precipitación y temperatura media con una precisión máxima de 10 km^2 desde 1950 hasta 2023. Estos datos también se proporcionan en formato netCDF y cubren una superficie de 3.278.250 km^2 con datos de 27.028 días en el periodo más largo, lo que se consideró suficiente para probar los límites del programa a desarrollar. Además, no solo proporciona todos los datos de ese periodo, sino que también se proporcionan en una menor precisión, 25 km^2, y en distintos periodos de tiempo de entre 9 y 15 años. Esto permite un desarrollo progresivo que facilita la depuración del código y una comprobación del funcionamiento más rápida y sencilla, por lo que finalmente se decidió escoger esta fuente de datos. Estos datos proceden tanto de *European National Meteorological and Hydrological Services* como de otras instituciones propias de cada país que colaboran con Europa.

2.4. Tecnologías empleadas

En esta Sección se mencionaran las tecnologías que han sido usadas durante el desarrollo de este TFG y se dará una perspectiva general de algunas tecnologías que se usan actualmente para la Computación de Alto Rendimiento(en inglés HPC).

2.4.1. Formato del conjunto de datos (*dataset*)

Como ya se comento en la Sección 2.3, tanto la AEMET como Copernicus proporcionaban los datos en formato netCDF [9]. El formato netCDF es un formato de tipo raster capaz de almacenar datos multidimensionales. Este formato, junto con HDF5, son dos de los formato más populares para almacenar grandes cantidades de datos científicos multidimensionales. Estos formatos comparten muchas capacidades, dado que netCDF4 utiliza HDF5 internamente [10]. Debido a que los datos encontrados se encontraban en formato netCDF4 y es un formato popular, se decidió utilizar este tipo de formato. Una ventaja de este formato es su librería. La librería de netCDF tiene mucha documentación y su rendimiento es suficiente para nuestra problemática. Además, la librería de netCDF permite leer archivos en formato HDF5, lo que ayuda a que el programa sea más compatible si en el futuro se decide utilizar con este tipo de formato. Se explicará más en detalle conceptos básicos de la estructura de un archivo netCDF y su librería en la Sección 2.6.

2.4.2. Lenguaje de Programación, librerías e interfaces

El lenguaje de programación elegido para desarrollar el programa fue C, ya que es ampliamente utilizado en HPC y uno de los lenguajes con mejor rendimiento [11]. Uno de los objetivos principales de este programa es aprovechar el paralelismo dentro del procesador. Para ello, se decidió utiliza la interfaz de programación de OpenMP [12]. OpenMP utiliza directivas *#pragma* para establecer zonas paralelas, paralelizar bucles y establecer distintas instrucciones típicas de la programación paralela como zonas críticas. OpenMP es ampliamente utilizado en la Computación de Alto Rendimiento [13] debido al gran rendimiento que proporciona y su facilidad de uso.

Para aprovechar la potencia de las tarjetas gráficas(GPU), debido al desarrollo del programa que se explicará en detalle en el Apartado 3, el principal problema donde se podía aprovechar dicha potencia era en el ordenamiento de vectores. Para ello, se decidió utilizar Thrust [14]. Thrust es una librería de C++ que implementa numerosos algoritmos de forma paralela para CUDA (Compute Unified Device Architecture) creada por Nvidia. Esta decisión se debió a que se consideró complicado mejorar una implementación de ordenamiento en GPU que mejorara a esta librería, y que los equipos utilizados para el desarrollo pertenecientes al Departamento de Arquitectura y Tecnología de Sistemas Informáticos (DATSI) tenían tarjetas gráficas Nvidia. Sin embargo, a pesar de ser una librería, ofrece funciones de bajo nivel cuyo uso es habitualmente necesario para obtener un rendimiento óptimo. Como consecuencia, es necesario conocer la arquitectura de CUDA, como se explicará en la Sección 2.5. Además, una de las peculiaridades de Thrust es que permite ejecutar el código tanto en el procesador como en la GPU, lo que resultará útil para optimizar la implementación como se verá en el capítulo de Desarrollo 3. También, se utilizaron librerías especializadas como netCDF para interactuar con el *dataset*, como se explicará en la Sección 2.6.

2.5. Arquitectura de CUDA

Una tarjeta gráfica está basada en el paralelismo masivo de datos. Esta formada por múltiples multiprocesadores y cada multiprocesador contiene una gran cantidad de núcleos (en inglés *cores*). Cada multiprocesador contiene sus propios bancos de registros, memoria compartida y caches. Cada núcleo equivaldría a una unidad aritmético lógica (ALU) que realizaría los cálculos necesarios. Para aprovechar esta arquitectura, es necesario desarrollar funciones llamadas *kernel* específicas para que ejecuten en la GPU. Estas funciones deben estar programadas para aprovechar una gran cantidad de hilos, llegando hasta los 1024 hilos. Antes de llamar a un kernel, es necesario copiar los datos necesario en la GPU, lo que supone un tiempo extra. Cuando llamas a una función *kernel*, es necesario indicar el número de bloques, que pueden ser miles, y el número de hilos que tendrá cada bloque, que pueden ser hasta 1024 hilos. Cada bloque contiene sus propias caches que permiten a los hilos comunicarse entre si. De forma simple, al llamar a una función *kernel*, en los bloques indicas el número de veces que se ejecutará esa función de forma paralela, y en el número de hilos se indica cuantos hilos ejecutarán una función de forma concurrente. Estas llamadas son asíncronas, pudiendo ejecutar código en el procesador mientras que la GPU termina su ejecución. Cuando el procesador necesite los datos, es necesario copiar desde la GPU a la CPU con una llamada bloqueante.

Con Thrust, se simplifican estas llamadas y no es necesario especificar el número de bloques ni el número de hilos. A partir de la versión 1.9.4 de Thrust, los desarrolladores decidieron hacer que todas las llamadas a Thrust fueran síncronas, ya que esto variaba dependiendo de las llamadas. Esto provoca que no se pueda paralelizar la ejecución del procesador con la de la GPU, bloqueando el procesador. Sin embargo, es posible ejecutarlo de forma asíncrona con el uso de streams.

2.5.1. Thrust asíncrono y *streams*

Un *stream* en CUDA es una secuencia de operaciones que ejecutan en el dispositivo en el orden en el que se van llamando en el código del procesador. Sin embargo, diferentes operaciones en distintos intervalos pueden alternarse o incluso ejecutar concurrentemente. Thrust permite ejecutar sin indicar ningún *stream*, ejecutando siempre en un *stream* por defecto. Esto provoca la desventaja de no poder ejecutar concurrentemente dos o más *kernels* en caso de ser posible. Además, los streams se pueden configurar para sobrescribir la sincronía de las llamadas de Thrust, permitiendo al procesador no solo ejecutar código mientras que la GPU termina de ejecutar, sino también poder hacer múltiples llamadas a Thrust. En el Apartado 3.4 se detallará como se ha realizado esto y en el Apartado 4.2 se mostrará si se ha conseguido mejorar los tiempos con su uso.

2.6. Estructura de los archivos netCDF y su librería

Como se describió brevemente en el Apartado 2.4.1, para el conjunto de datos utilizaremos el formato netCDF. Un archivo netCDF contiene dos elementos esenciales, las dimensiones y las variables. Las dimensiones definen la forma que tienen los datos en el archivo, y tienen un nombre y un tamaño asociado. Las variables contienen los propios datos. Las variables tienen un nombre, un tipo y la forma en la que están guardados los datos. Esta forma está descrita con las dimensiones, que permiten dar la forma multidimensional a las variables. Esto permite ciertas optimizaciones a la hora de acceder a los datos, aunque también tiene sus desventajas.

Tal y como explica Russ Rew, desarrollador en Unidata [15], organización propietaria de netCDF, en su post en el blog para desarrolladores de Unidata [16], los patrones de acceso en un archivo netCDF puede modificar el tiempo de lectura notoriamente.
Partiendo del ejemplo de la figura 2.1, donde el parámetro *time* representa la dimensión que varía más despacio, el parámetro *y* la siguiente dimensión más rápida en variar, y el parámetro *x* la dimensión más rápida en variar, si queremos acceder de la forma representada a la izquierda, el tiempo de lectura será de aproximadamente 180 segundos, 14.000 veces más lento que si accedemos de la forma representada a la derecha, que tardaría aproximadamente 0.013 segundos. Debido a esto, si queremos realizar el acceso de la forma de la izquierda, sería necesario reorganizar los datos, haciendo la dimensión más rápida en variar el tiempo. En el caso de que interesen ambas formas de acceso, es posible dividir el *dataset* en trozos (*chunking*), haciendo el acceso más lento más rápido, a costa de hacer el acceso más rápido más lento.

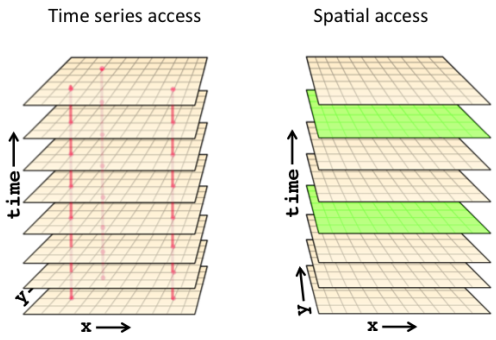

Figura 2.1: Tipos de acceso en un archivo netCDF. [16]

Con la librería netCDF, podemos leer toda la información sobre las dimensiones y las variables. Primero, es necesario abrir el archivo con *nc_open*. Una de las ventajas de esta librería es que esta preparada para que el archivo pueda ser accesible a través de distintas máquinas utilizando MPI (interfaz de paso de mensajes)[17]. Si solamente queremos acceder a una parte de una variable multidimensional, ya que en mucho casos no será posible leer toda una variable multidimensional debido a la falta de memoria, primero es necesario conocer la forma en la que están almacenados los datos en las variables. Debido a que es posible que no se conozcan, existen llamadas para iterar por todas las dimensiones o variables disponibles, como se puede ver en el Listing 2.1.

Listing 2.1: Accesos a datos con netCDF

```
//Se extrae el número de dimensiones(ndimsp) y variables(nvarsp)
```

```
//que hay en el fichero abierto(ncidp)
nc_inq(ncidp, &ndimsp, &nvarsp, &nglattrsp,&unlimdimidp);
//Iteramos por cada dimension sacando su nombre(nameDim)
for(i=0; i<ndimsp; i++)
 nc_inq_dim(ncidp,i, nameDim, &ldimsp);
//Iteramos por cada variable sacando su nombre(nameVar),
//tipo(varType) y el id de las dimensiones(dimId).
//DimId está ordenado desde la que varía más despacio
//hasta la que varía más deprisa
for(i=0; i<nvarsp; i++)
 nc_inq_var(ncidp,i,nameVar,&varType,&ndimsp,dimId,&nglattrsp);
```

Las dimensiones normalmente contienen nombres descriptivos para indicar las forma en la que están almacenadas las variables. Una vez se conocen dimensiones, ya se pueden extraer los datos de las variables de la forma que interese. Además, netCDF tiene publicados ejemplos sencillos para la escritura y la lectura de dimensiones y variables [18]. Se desarrollará más sobre esto en el capítulo de Desarrollo 3.

Capítulo 3

Desarrollo

En este capítulo se detallarán todos los pasos que se han realizado para conseguir crear el programa de cálculo de la anomalía climática y conseguir buenos rendimientos. Todo el código de las implementaciones se puede consultar en github [19].

3.1. Análisis de *datasets*

Como se explicó en la sección 2.6, la forma en la que los datos estén distribuidos en una variables es esencial para obtener tiempos de lectura buenos. Por ello, en este apartado se va a explicar la estructura de los datos de Copernicus, y analizar los propios datos para asegurar que son correctos.

3.1.1. Estructura de los *datasets*

Dado que en Copernicus se puede seleccionar distintos periodos y precisiones, se decidió escoger tres conjuntos de datos. Un *dataset* pequeño, con una resolución de 0,25 grados, equivalentes a 25 km^2, con un periodo de 9 años, un *dataset* mediano, con una resolución de 0.1 grados, equivalentes a 25 km^2, con un periodo de 15 años, y un *dataset* grande con resolución 0.1 grados y todo el periodo disponible, 73 años. Al descargar los datos, cada medición venía en un fichero distinto, teniendo dos *datasets*, uno para la temperatura media y otro para la precipitación. Estos *datasets* no contenían ningún fichero asociado con información relativa a ellos , a diferencia de los *datasets* de la AEMET, que si contenían un fichero indicando la distinta información sobre ellos como nombre y tamaño de dimensiones y el tipo, nombre y la forma de las variables sobre las dimensiones [2.1]. Debido a este hecho, fue necesario implementar un programa sencillo capaz de extraer la información básica de los *datasets* parecido al descrito en el Listing 2.1.

Una vez implementado, se pudo observar que ambos *datasets* contenían tres dimensiones y 4 variables. Las dimensiones correspondían a la latitud, longitud y los días, que también tenían sus respectivas variables para almacenar su

propio valores. En el caso de las variables latitud, longitud y días están relacionadas solamente con su dimensión equivalente, conformando un solo vector de datos. En cambio, para la cuarta variable del *dataset*, ya sea la temperatura o la precipitación dependiendo del conjunto de datos que se lea, están relacionadas con las tres dimensiones, siendo la dimensión tiempo es la que varía más lentamente, seguida por la dimensión latitud que varía más rápidamente y la dimensión longitud siendo la que varía más rápido. Para el cálculo de la anomalía climática, es necesario conocer todos los datos diarios para una determinada posición. Esta casuística provocó que fuera necesario implementar un programa que reordenara estas variables, de forma que el tiempo fuera la que varíe más rápido, tal y como se menciono en la sección 2.6.

3.1.2. Análisis, Filtrado e Interpolación de datos y Transformación de *dataset*

Tras conocer la estructura de los *datasets*, se decidió implementar un programa para revisar que los datos eran coherentes, además de para familiarizarse con la librería netCDF. Este programa leía los valores de latitud, longitud y precipitación o temperatura de cada *dataset*, y los recorría todos buscando los valores máximos y mínimos de la latitud y longitud, y la media y el número de datos inválidos de la precipitación y la temperatura. Tras la primera ejecución, se observó que aproximadamente un 80 % de los datos de precipitación eran inválidos. Con los valores máximos y mínimos de latitud y longitud se pudo trazar la superficie que cubrían los datos, mostrando que gran parte de los puntos se encontraban en el océano, lo que explicaría el gran número de inválidos. Por ello, se amplió el programa, que ahora buscaría además la máxima y mínima latitud y longitud que contenían algún valor válido. En la figura 3.1, se puede ver en color rojo el perímetro de la superficie que cubría todo el *dataset*, y en morado el perímetro de la superficie conformada por la máxima y mínima latitud y longitud que contenían algún valor válido.

En consecuencia, y sumado a la necesidad de transformar el *dataset* para una óptima lectura, se decidió implementar un programa capaz de reducir las coordenadas de latitud y longitud a las realmente efectivas al mismo tiempo que reordenaba las variables precipitación y temperatura creando un nuevo conjunto de datos. Además, se añadió la capacidad de interpolar los valores que sean nulos dentro del área efectiva.

La primera versión de dicho programa primero leía todos los valores de longitud, latitud y precipitación y temperatura. Esto no resultará válido para el ejemplo más grande y complicará notablemente la implementación, como se explicará en el Apartado 3.5. Después, se comprobaba si cada latitud era válida. Una latitud se consideró valida si contenía al menos un 5 % del máximo de valores válidos. Por tanto, se debe recorrer todos los valores de cada longitud para cada día comprobando si la precipitación era mayor que 0 y menor que 300 mm3. La razón de poner una límite relativamente bajo fue para intentar evitar que las latitudes que estén conformadas mayormente por agua pero contengan también tierra no sean descartadas. Tras esta comprobación se hace lo mismo pero para la longi-

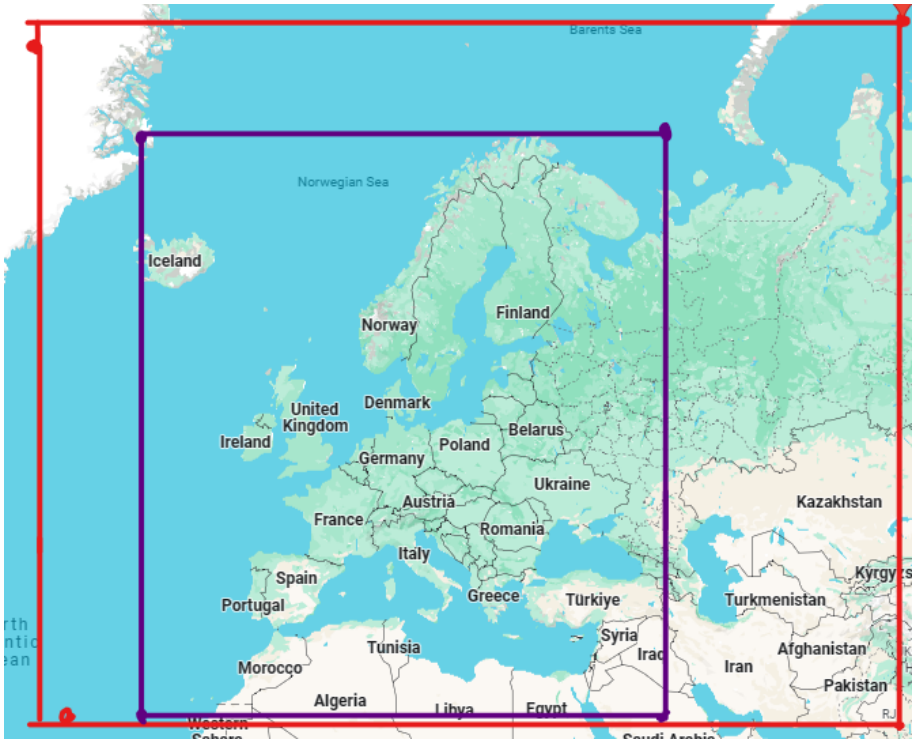

Figura 3.1: Área efectiva con datos en Europa

tud. Al finalizar las comprobaciones, se empiezan a realizar las interpolaciones y la escritura en el nuevo *dataset* optimizado para el cálculo de la anomalía climática.

Al tener cargados todos los valores en memoria, la escritura e interpolación resultó relativamente sencilla. Para la escritura, fue necesario declarar los nuevos ficheros las variables precipitación y temperatura con la forma adecuada para su lectura óptima para el cálculo de la anomalía climática, como se puede ver en el Listing 3.1. Además, se decidió cambiar el tipo de la variable precipitación, que era tipo *double*, a tipo *float*, debido a que se consideró que el extra de precisión no resultaría útil en el cálculo, y se reducirían el uso de memoria y el tamaño del *dataset*.

Listing 3.1: Accesos a datos con netCDF

```
//Creamos el fichero guardando el id (newNcId)
nc_create("copernicus_tg_data_clean.nc", NC_CLOBBER, &newNcId);
//Creamos las dimensiones guardando sus id (new***Dim)
nc_def_dim(newNcId, "time", TIME, &newTimeDim);
nc_def_dim(newNcId, "longitude",newLonSize, &newLonDim);
nc_def_dim(newNcId, "latitude", newLatSize, &newLatDim);
//Almacenamos en un array los ids de las dimensiones para,
//la creación de las variables con la forma correcta, siendo
//el primer valor el que variará más lento, y el último el
//que variará más deprisa
dimIds[0]=newLatDim;
```

```
dimIds[1]=newLonDim;
dimIds[2]=newTimeDim;
//Definimos las variables unidimensionales
nc_def_var(newNcId, "longitude", NC_DOUBLE, 1, &newLonDim,
                                                    &newLonId);
nc_def_var(newNcId, "latitude", NC_DOUBLE, 1, &newLatDim,
                                                    &newLatId);
nc_def_var(newNcId, "time", NC_DOUBLE, 1, &newTimeDim,
                                                    &newTimeId);
//Definimos la variable multidimensional temperatura
nc_def_var(newNcId, "tg", NC_SHORT, 3, dimIds, &newTgId);
```

En la escritura del fichero, es necesario que todos los valores escritos correspondan con la forma que se ha definido en la creación del fichero para resultados correctos. Al tener cargados los valores en memoria, es posible recorrerlos directamente de esta forma para una escritura sencilla. La interpolación se realizó de forma muy simple. En caso de que una posición tenga un valor nulo, se mira a los valores que se encuentran a su alrededor y escoge uno que no sea válido. Esta interpolación resolvió algunas situaciones, pero resulto insuficiente, provocando que durante el cálculo fuera necesario sustituir valores inválidos por valores medios. Finalmente, al ejecutar por primera vez con los *datasets* pequeños y medianos, se observó que los *datasets* de precipitación contenían un par de latitudes y longitudes menos que los *datasets* de temperatura, por lo que para mantener la consistencia entre los *datasets* se decidió mantener para ambos las latitudes y longitudes válidas del *dataset* de precipitación. Sin embargo, se dejó el código necesario para evaluar por separado cada *dataset*. Una vez creados los nuevos ficheros, se realizó nuevamente el análisis de datos y, tanto para los medianos como para los pequeños, se mostró que la cantidad de valores inválidos se había reducido a un 20 % en los *datasets* de precipitación y a un 7 % en los *datasets* de temperatura. Seguidamente, con los *datasets* con una cantidad de valores inválidos razonable y optimizados para el cálculo de la anomalía climática, se prosiguió con la implementación del programa encargado de realizar dicho cálculo.

3.2. Primera implementación secuencial y profiling

3.2.1. Implementación Secuencial

Para sintetizar los objetivos de la implementación, el programa debe calcular la anomalía climática, y por tanto, como se explicó en el Apartado 2.1, debe calcular el *drought code* del índice canadiense[3] necesario para calcular el estrés hídrico, el propio estrés hídrico, el estrés térmico, y finalmente la anomalía climática. Esta implementación, a pesar de ser la secuencial, ya se escribió pensando en las distintas posibilidades de paralelización y en el procesamiento de grandes cantidades de datos. Por ello, se decidió que sería más óptimo para evitar llenar la memoria realizar todos los cálculos necesario para calcular la anomalía climática de un solo punto, permitiendo reservar mínimo tamaño

en memoria posible, antes de calcular la siguiente posición. Este tamaño sería el número de días que se quisieran calcular. Por tanto, se realizan dos bucles anidados, siendo el bucle exterior la latitud, y el bucle interior la longitud, y después leyendo la precipitación y temperatura de todos los días de ese punto en una sola llamada.

Tras la lectura, se calcula el *drought code* para cada uno de los días. El código implementado utiliza la biblioteca math.h para realizar los cálculos necesarios, explicados en la página *Fire Weather Indices* del Instituto Federal Suizo de Investigación[3]. Debido a que es el cálculo del *drought code* depende del valor del día anterior para calcularse, en caso de que haya un valor inválido en la precipitación se sustituye por un 0, y si hay un valor inválido en la temperatura se sustituye por el valor medio de la temperatura que se ha observado en el apartado de análisis. Además, se ha añadido un límite al número de valores que se pueden sustituir, el 30 % de los valores máximos. Estos cálculos se realizaban en una función que en caso de superar el límite, retornaba 1, provocando que se establecieran el estrés hídrico, el estrés térmico y la anomalía climática como -1 para todos los días de esa posición. Esta función se puede ver en el Listing 3.2;

Listing 3.2: Accesos a datos con netCDF

```
for(i=1;i<TIME;i++){ //Iteramos por todos los días del punto
  invalid=0;
  if(*(rainfall+i) < 0.0 || *(rainfall+i) > 300.0){
  //Si no se tiene datos, se asume que no ha llovido
    *(rainfall+i)=0;
    invalid=1;
  }
  if(*(temperature+i) < -8000 || *(temperature+i) > 8000){
    *(temperature+i) = 1040; //Temperatura media del dataset
    invalid=1;
  }
 if(invalid)
  totalInvalid++; //Aumentamos el contador de inválidos
 if(*(rainfall+i)>2.8){ //Función de cálculo del drought code
   efRainfall = 0.86 * *(rainfall+i) -1.27;
   prevMoisture = 800 * exp(-*(droughtCode+i)/400);
   moisture = prevMoisture + 3.937 * efRainfall;
   prevDroughtCode = 400 * log(800/moisture);
   prevDroughtCode = prevDroughtCode < 0 ? 0 : prevDroughtCode;
 }
 evapotranspiration = 0.36 * (*temperature + 2.0)
                              + getDayLength(*(days+i));
 evapotranspiration = evapotranspiration < 0 ? 0
                              : evapotranspiration;
 *(droughtCode+i) = prevDroughtCode + 0.5 * evapotranspiration;
}
if(limite < totalInvalid) //Si se supera el límite da error
 return 1;
```

```
return 0;
```

Posteriormente, en caso de que el *drought code* se hubiera calculado con éxito, se calculaba el estrés hídrico y térmico. Para este cálculo, primero es necesario crear los percentiles para dicha posición. Para ello, primero es necesario ordenar todos los valores de menor a mayor, y se utilizó el método de ordenamiento qsort de la librería estándar de C para conseguirlo. Una vez se tenían los valores ordenados, se colocaban en un array los 99 valores que dividían los percentiles y se buscaba a que percentil pertenecía cada valor mediante una ligera variación del algoritmo *binary search*. Esta variación se debe a que estamos buscando si un valor pertenece a cierto rango y no el valor exacto. Esta variación se basa en que cuando se alcanza la condición de parada del *binary search* si no se encuentra el valor, en este caso el valor pertenece al percentil anterior que se ha consultado. El código puede verse en el Listing 3.3, donde en la variable percentiles se pasa el array con los valores que dividen cada percentil y en val el valor del que se quiere encontrar el percentil al que pertenece. Tanto el estrés hídrico como el estrés térmico se almacenaron en variables de tipo short, ya que solamente pueden contener valores del 1 al 99 correspondientes al percentil al que pertenecen.

Listing 3.3: Variación del binary search

```
int encontrarPercentilDouble(double *percentiles, double val){
  int last_percentil=0, first=0, middle=49, last=99;
  while(first<=last){
    if(*(percentiles+middle) < val){
//Guardamos el percentil inferior al que petenecería el valor
      last_percentil=middle;
      first = middle +1;
    } else if(*(percentiles+middle)== val){
      last_percentil = middle;
      break;
    } else {
      last = middle -1;
    }
    middle = (first +last) /2;
  }
  //+1 porque la posición del array empieza en 0
  return last_percentil +1;
```

Finalmente, se calcula la anomalía climática haciendo la media entre el estrés hídrico y el estrés térmico y se escribe en un nuevo fichero para la anomalía climática. Este fichero se creó con la variable *'anomaliaClimatica'* manteniendo la misma forma que los fichero optimizados para el cálculo, es decir, manteniendo la dimensión de los días como la que varía más deprisa. Esta decisión se debe a que es posible que este fichero se utiliza en la posteridad para el cálculo del Índice de Propagación Potencial.

3.2.2. Análisis de rendimiento del programa secuencial

El análisis de rendimiento del programa, o *profiling*, con herramientas especializadas permite visualizar el comportamiento de un programa en distintos campos, además de permitir la depuración de errores más detalladamente. Como herramienta de *profiling* se utilizó Valgrind [20], que contiene a su vez algunas herramientas específicas propias de cada problemática. Valgrind detecta los errores de memoria, analiza los accesos a caché con la herramienta Cachegrind [21], detectando tanto aciertos como fallos, y genera un grafo de llamadas con Callgrind [22], en el que muestra el coste de cada parte del código, es decir, el tiempo relativo que se están ejecutando las instrucciones respecto al tiempo total del programa, entre otras funcionalidades.

Tras ejecutar, los resultados, como se puede observar en la Figura 3.2, sobre los accesos a caché fueron que, de un total de 27.208 millones de accesos, 132 mil fueron fallos, un 0.5 % del total de accesos. Estos resultados se deben tanto a la implementación, que se realizó pensando en reducir estos fallos, por lo que siempre se accede a valores contiguos en memoria, y a la optimización del *dataset* que se realizó en el apartado de análisis del *dataset*.

```
==387951==
==387951== I   refs:      93,525,008,088
==387951== I1  misses:             9,053
==387951== LLi misses:             6,202
==387951== I1  miss rate:           0.00%
==387951== LLi miss rate:           0.00%
==387951==
==387951== D   refs:      27,208,664,971  (18,641,083,032 rd   + 8,567,581,939 wr)
==387951== D1  misses:       132,756,372  (    79,246,765 rd   +    53,509,607 wr)
==387951== LLd misses:            54,885  (        31,678 rd   +        23,207 wr)
==387951== D1  miss rate:            0.5% (           0.4%     +           0.6%  )
==387951== LLd miss rate:            0.0% (           0.0%     +           0.0%  )
==387951==
==387951== LL refs:          132,765,425  (    79,255,818 rd   +    53,509,607 wr)
==387951== LL misses:             61,087  (        37,880 rd   +        23,207 wr)
==387951== LL miss rate:             0.0% (           0.0%     +           0.0%  )
```

Figura 3.2: Accesos a cache analizados con Cachegrind.

En el grafo de llamadas representado en la Figura 3.3, se observa que las funciones de estrés térmico y estrés hídrico eran en las que el programa estaba alrededor de un 80 % del tiempo de ejecución. Dentro de estas funciones, la función qsort de la librería estándar de C es la que estaba alrededor del 70 % del tiempo de todo el programa. En el resto de funciones, como el cálculo del *drought code* o la búsqueda del percentil al que pertenecían los valores tardaban entre un 4 % y un 7 %. Esta información llevó a la conclusión de que la principal función que se debía introducir en la GPU para intentar mejorar tiempos sería el ordenamiento.

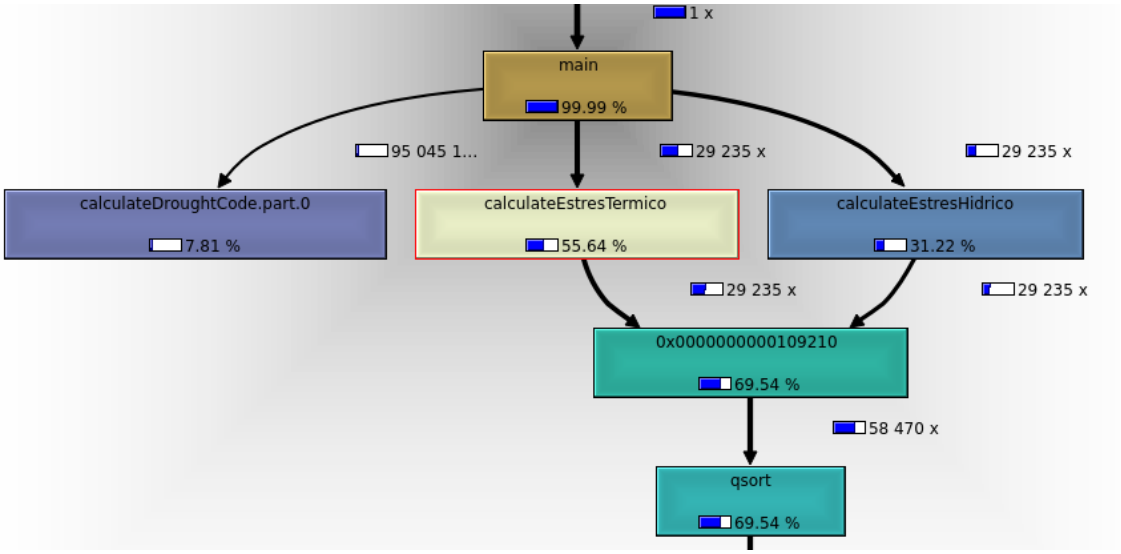

Figura 3.3: Grafo de llamadas con costes.

3.3. Implementación paralela en CPU con OpenMP

La implementación paralela en CPU resultó ser fácilmente adaptable, ya que durante la programación secuencial ya se pensó en ella, para evitar que la transición resultará complicada. Para paralelizar el código se utilizó la interfaz de programación de OpenMP [12], como ya se explico en el Apartado 2.4.2.

3.3.1. Paralelización con OpenMP

Para paralelizar el programa, se decidió crear la zona paralela después de abrir los archivos de los *datasets*, pero antes de la reservas de memoria con la directiva del Listing 3.4. Esta directiva inicializa los hilos y delimita la zona que debe ser paralela mediante corchetes.

Listing 3.4: Directiva para inicializar los hilos

```
#pragma omp parallel
```

Al realizar las reservas de memoria dentro de la zona paralela, cada hilo contiene su propia memoria exclusiva, y solamente reserva la necesaria para calcular la anomalía climática de un solo punto. Esto no produce problemas con la cantidad de memoria reservada ya que la cantidad de memoria necesaria para una solo punto no supera el megabyte ni en el ejemplo más grande. Después, con la directiva del Listing 3.5, OpenMP es capaz de distribuir las iteraciones de un bucle entre los hilos, de forma que cada hilo ejecuta de forma independiente cada iteración. De esta forma, cada hilo ejecutará todos los cálculos respecto al punto que se le ha asignado de forma independiente sin tener que esperar a otros hilos.

Listing 3.5: Directiva para distribuir bucle entre hilos

```
#pragma omp for schedule(guided)
```

La anotación *schedule* permite indicar a OpenMP como distribuir las iteraciones de un bucle entre los hilos. Existen tres tipos de distribuciones: *Static*, que distribuirá el mismo número de iteraciones entre los hilos. Como ejemplo, en un bucle con 100 iteraciones y dos hilos, el primer hilo se encargaría de las primeras 50 y el segundo de las últimas 50; *(Dynamic,chunkSize)*, en la que se le indica en *chunkSize* el número de iteraciones que debe realizar un hilo antes de pedir más iteraciones. De esta forma, si hay iteraciones que son más costosas que otras como ocurre en el cálculo del *drought code*, donde si la precipitación es menor a 2.8 mm realizamos menos cálculos, evitaríamos tener hilos ociosos esperando a otros hilos como podría ocurrir con la distribución *Static*; La última distribución es *Guided*, que sigue el mismo paradigma que la distribución *Dynamic*, pero el *chunkSize* varía durante la ejecución del programa. Debido al rendimiento de cada distribución, que se detallarán más adelante en el Apartado de Resultados con OpenMP 4.1, se decidió elegir *Guided*.

3.3.2. Corrección de Errores

Tras la primera ejecución, se observaron errores en la lectura de los valores. Esto se debía a que múltiples hilos estaban intentando leer simultáneamente a través del mismo descriptor de fichero distintas zonas de memoria. Una de las posibles soluciones fue establecer la directiva *#pragma omp critical* en la lectura de los datos, pero esto provocaría que algunos hilos tuvieran que esperar. La solución que se encontró fue abrir el fichero tantas veces como hilos se iban a crear. De esta forma, cada hilo tendría su propio descriptor para leer. Sin embargo, esto obligaba a establecer el número de hilos de forma estática dentro del código. Se estableció la constante *NUM_THREADS* con *#DEFINE*, y a continuación de la directiva del Listing 3.4, se añadió la anotación *num_threads(NUM_THREADS)*, que permite indicar el número de hilos que debe arrancar. Tras esta corrección, los tiempos se redujeron notablemente, como se describirá en el Apartado Resultados con OpenMP 4.1.

3.4. Implementación paralela en CPU con OpenMP y GPU con CUDA

En esta implementación se busco mejorar la implementación descrita en el apartado anterior con el uso de las tarjetas gráficas para el ordenamiento, que como se describió en la Sección 3.2.2, era la función que se llevaba más tiempo del programa.

3.4.1. Utilización de Thrust

Dado que implementar un algoritmo de ordenamiento en GPU no resulta algo trivial y CUDA tiene librerías especializadas, se consideró difícil mejorar los algoritmos que proporcionaban la librerías de Nvidia, y por ello se decidió utilizar Thrust[14], como se mencionó en el Apartado 3.4. Para comprobar el rendimiento de esta librería, se decidió realizar un pequeño fichero con pruebas simples

que simularán nuestro problema para comparar de forma sencilla los rendimientos entre la CPU y la GPU, y que además comparara el rendimiento del algoritmo de ordenamiento qsort de la librería estándar de C contra Thrust en CPU.

3.4.2. Comparación de Métodos y Primera Implementación

La problemática fue ordenar un array de máximo 27.028 elementos, que representa el ejemplo más grande. Se entrará más en detalle en el Apartado 4.2, pero se observó que el ordenamiento en Thrust tardaba cerca de 10 veces menos en procesador(CPU) comparado con el qsort de la librería de C. Esta diferencia resultaba en una gran mejora de tiempo en el programa, cercana a la mitad. Sin embargo, los tiempos en GPU fueron peores. El uso de la GPU se reservó al ordenamiento de la temperatura, mientras que la CPU calcularía el *drought code* y el estrés hídrico. En un principio, al ser las llamadas a GPU asíncronas, esta versión se beneficiaría del paralelismo entre GPU y CPU. Se pensó que podía deberse a que el tamaño del array era pequeño comparado con la capacidad de procesamiento de una GPU y, con el coste añadido de enviar los datos a la GPU, resultaba en ligeramente peores tiempos que la CPU. Por ello, se buscó una solución capaz de ordenar un array de muchos elementos, como podría ser la combinación de para una latitud con todas sus longitudes para sus respectivos días. Esto equivaldría a 592 longitudes en el ejemplo grande, aumentando el array de 27028 a 16 millones de elementos. La principal dificultad de esta implementación es separar los arrays de cada longitud a pesar de ordenarlos todos juntos.

3.4.3. Implementación *Back-To-Back*

Tras investigar las posibles soluciones, se encontró una solución conocida como *Back-To-Back*. La solución crea un array extra de claves que indican a que array pertenece cada valor. Después se ordena todo el array de forma normal, desordenando las claves. Una vez todos los valores están ordenados, se reordenan de forma estable, es decir manteniendo el orden en caso de que los valores sean iguales, siguiendo las claves. De esta forma, las claves que tengan el mismo valor acabarán juntas, pero al ser un ordenamiento estables, mantendrán el orden que tenían al ser ordenadas en conjunto. En el Listing 3.6 se puede ver la implementación y un ejemplo sencillo del ordenamiento.

Listing 3.6: Ordenamiento Back-to-Back con thrust

```
// Ejemplo de funcionamiento
// -6 2 4 1 3 5 ==> -6 1 2 3 4 5 ==> -6 2 4 1 3 5
//  0 0 0 1 1 1 ==>  0 1 0 1 0 1 ==>  0 0 0 1 1 1

//Auxiliar que guarda las claves.
thrust::device_vector<float> d_keys(LON * TIME);
//Rellena las claves.
thrust::transform(thrust::make_counting_iterator(0),
                  thrust::make_counting_iterator(LON * TIME),
                  thrust::make_constant_iterator(LON),
```

```
                    d_keys.begin(),
                    thrust::divides<int>());
// --- Back-to-back approach
//Ordenamos todos los elementos del vector
thrust::stable_sort_by_key(device_part_tg_val,
                           device_part_tg_val + LON * TIME,
                           d_keys.begin(),
                           thrust::less<float>());
//Ordenamos las claves.
thrust::stable_sort_by_key(d_keys.begin(),
                                  d_keys.end(),
                                  device_part_tg_val,
                                  thrust::less<int>());
```

Esta implementación, al igual que la anterior, solo ordenaba la temperatura en GPU con el método anterior, mientras la CPU calculaba *drought code* y el estrés hídrico. Hay que destacar que esta implementación consume tantas veces más memoria como longitudes tengamos que almacenar para realizar el cálculo anteriormente descrito, tanto para la temperatura como para la precipitación, lo que puede resultar en un problema de memoria, aunque en este caso, para el ejemplo más grande se reservarían alrededor de 350 megabytes de memoria por hilo de ejecución, lo que no supone un problema. A pesar de esto, los tiempos, aunque mejores, seguían siendo peores que solo utilizando la CPU, a pesar del supuesto paralelismo entre CPU y GPU. Esto llevo a realizar una investigación más profunda de Thrust, y como resultado, se descubrió que a partir de la versión 1.9.4 de Thrust, todas las llamadas eran síncronas, bloqueando los procesos de la CPU, lo que explicaría porque los tiempos resultaron peores.

3.4.4. Implementaciones asíncronas

Como consecuencia, se realizaron dos versiones asíncronas utilizando *streams*, con las ventajas que se mencionaron en el Apartado 2.5.1. La creación de los *streams* y la política de ejecución asíncrona sobre cada *stream* pueden verse en el Listing 3.7.

Listing 3.7: Creación de Streams y política de ejecución Asíncrona

```
//Se crean antes de la zona paralela
cudaStream_t streams[NUM_THREADS];
for(i=0; i<NUM_THREADS; i++)
 cudaStreamCreate(&streams[i]);
//Se inicia la zona paralela
auto nosync_exec_policy =
                thrust::cuda::par_nosync.on(streams[threadId]);
//Ejemplo de llamada con la política creada
thrust::stable_sort_by_key(nosync_exec_policy,d_keys[threadId],
                                 d_keys[threadId]+LON*TIME,
                                 device_part_tg_val[threadId],
                                 thrust::less<short>());
```

La versión en la que se realizaba una llamada para ordenar el array de un punto al igual que en la versión paralela del procesador (apodada como *ManyCalls*), debería presentar alguna ventajas, ya que si la implementación es correcta se podrían ejecutar en paralelo varías llamadas al utilizar distintos *streams*. La versión Back-To-Back se beneficiaría especialmente de la asincronía al ser una ejecución más pesada. Sin embargo, los tiempos no mejoraron y solo consiguieron igualar a la versión paralela solo ejecutando en CPU en el mejor de los casos.

Las razones que se contemplaron para este mal funcionamiento fueron tanto el mal aprovechamiento de los *streams*, que depende de la implementación de Thrust y es desconocida, y de la asincronía, ya que si las llamadas fueran realmente asíncronas, aunque no se aprovecharan correctamente los *streams*, los tiempos deberían mejorar, al menos ligeramente, al aprovechar el paralelismo en CPU y GPU. Tras revisar específicamente la política creada, resulta que solamente devuelve a ciertos algoritmos la asincronía que habían perdido con la nueva versión, como por ejemplo *thrust.sort* utilizado en la implementación *ManyCalls*. Sin embargo, los métodos de la implementación Back-To-Back continuaban siendo síncronos, concretamente los *sort_by_key*. Esto se debe a que este método utiliza memoria extra para ejecutar y la reserva desde la CPU antes de llamar al *kernel*, evitando que pueda ser asíncrono. Según foros de dudas de Nvidia, una posible solución sería crear *memory custom_allocator*, que permitiría hacer una reserva previa de memoria. De esta forma, estos métodos utilizarían esa memoria reservada en vez de reservarla dentro del método y evitando esa sincronía. Sin embargo, por falta de tiempo no se pudo implementar y los tiempos resultaron empeorar en el ejemplo grande como se verá en el Apartado 4.2.

3.5. Implementación final con ejemplos grandes

La diferencia de tamaño entre el ejemplo mediano y el ejemplo grande es el salto en la cantidad de datos que se deben tratar. En ambos ejemplos se tratan la misma cantidad de puntos, 437 latitudes y 592 longitudes, dando un total de 258.704 puntos distintos. La diferencia se encuentra en la cantidad de días de ambos ejemplos, 5.844 días (16 años) en el mediano frente a 27.028 días (74 años) en el grande, dando un total de datos de 1.511 millones frente 6.992 millones respectivamente. Esta diferencia provoca que el programa de filtrado y transformación de datos no sea válido para este ejemplo, limitado por la memoria del sistema. Por otro lado, la implementación del programa del cálculo de la anomalía climática ya se realizó pensando en el peor caso posible, por lo que su funcionamiento resulto correcto en este ejemplo.

3.5.1. Optimización del programa de Filtrado y Transformación de Datos

La primera implementación de este programa leía todas las variables del *dataset* a analizar, lo que facilitaba tanto el análisis de las latitudes y longitudes como la escritura en el nuevo *dataset* optimizado. Al tener que leer las variables parcialmente, surge el gran problema del acceso a las variables del *dataset*. Como se

ha comentado con anterioridad, en estos *datasets* la dimensión que varía más lento es el tiempo. Esto supone un problema tanto en la comprobación de las latitudes y las longitudes, como en la escritura de los datos en el nuevo *dataset*, ya que en ambos casos se necesita acceder a todos los valores de un punto tanto para comprobar si es válido como para interpolar sus valores y escribirlo en el *dataset*.

Para la comprobación de la validez de una latitud o longitud, la solución que se diseño fue realizar un array extra con el tamaño del número de latitudes o longitudes donde almacenar el número de puntos que eran inválidos para una latitud o longitud. De esta forma, es posible recorrer el *dataset* accediendo de forma diaria, almacenando en cada iteración el recuento parcial de puntos inválidos para cada latitud para dicho día. Al finalizar, se realiza otro bucle donde se comprueban en el array extra que latitudes o longitudes han superado el límite y no se incluyen en el nuevo *dataset*. En el Listing 3.8 se puede ver el código para la comprobación de las latitudes de la precipitación, equivalente para las longitudes intercambiando los bucles de recorrido dentro del bucle *TIME*, que siempre debe ser el externo para acceder de forma rápida al *dataset* como se explicó en el apartado 2.6.

Listing 3.8: Comprobación de latitudes para la precipitación

```
//Countp es la cantidad de datos que se leerán en cada lectura
//Startp es por donde se empezará a leer el dataset
*countp=1; //Se lee un dia
*(countp+1)=1; //Se lee una latitud
*(countp+2)=LON; //Se leen todos los datos de LON
*(startp+2)=0;//Se lee desde la primera longitud
//Se accede por día ya que es la dimensión que varía más lento
for(i=0;i<TIME;i++){
 *(startp)= i; //Se empieza a leer desde el día de la iteración
 for(j=0;j<LAT;j++){
   //Se lee los datos de todas las longitudes para la latitud
   *(startp+1) = j;
   nc_get_vara_double(ncIdPrec,precId,startp,countp,partPrecVal)
   for(k=0; k<LON;k++){
    //Si es inválido se incrementa el valor del array extra
     if(*(partPrecVal + k) < 0.0 || *(partPrecVal + k) > 300.0)
         *(notValid+j) = *(notValid+j)+1;
   }
 }
}
//Se comprueba si se ha superado el límite
for(i=0;i<LAT;i++){
//Si no se supera, se escribe al nuevo dataset
 if(*(notValid+i) < limite){
    *(newLat+*(newLatSize)) = *(lat+i);
    *(newLatSize) = *(newLatSize) +1;
 } else { //Si se supera, se establece como nulo
```

```
    *(lat+i) = -9999.0;
 }
}
```

Para la escritura de los datos en el *dataset*, se decidió diseñar una solución en la que mediante una constante se pudiera especificar en cuantas partes se debía dividir el *dataset* para la escritura. Esta solución dificultaba notablemente el código, pero requería mucha menos memoria y ejecutaba de forma más rápida. Se decidió dividir la escritura por latitudes. Primero, se reservaba la memoria necesaria para escribir en el nuevo *dataset* el trozo que se hubiera indicado mediante la contante. Posteriormente se realizaban las iteraciones correspondiente a ese trozo, y cada punto almacenaba sus valores en la parte del array que le correspondiera. Esto constituía la dificultad de que no todas las latitudes eran válidas dependiendo de las comprobaciones anteriores. En el Listing 3.9 el código esencial para la escritura. En este código también se realizaba la interpolación.

Listing 3.9: Escritura del nuevo dataset

```
//NUM_ESCR_PREC es la constante que define en cuantas partes
//se divide el dataset
//Iteraciones en cada Escritura
int numLatPerIter= LAT/NUM_ESCR_PREC;
//Reserva de memoria para la escritura de una parte del dataset
escrPrecVal=malloc(sizeof(float)*
          ((numLatPerIter+LAT%NUM_ESCR_PREC)*newLonSize*TIME));
for(z=0; z<NUM_ESCR_PREC; z++){ //Se iteran por las partes
 utilLatToWrite=0; //Cuantas latitudes son útiles de este trozo
 lastLatThisIter=numLatPerIter*(z+1);//Ultima LAT de esta iter
 if(z==NUM_ESCR_PREC-1){//Si es la ultima se hacen las sobrantes
  lastLatThisIter=LAT;
 }
 for(i=0; i<TIME; i++){
  *startp=i; //Se leen los datos de ese día
  utilLatToWrite=0;
  //Marcamos las latitudes que tocan esta iter
  for(j=z*numLatPerIter; j<lastLatThisIter; j++){
   if(*(oldLat+j) == -9999.0) continue; //No se escribe
   utilLatToWrite++;
   *(startp+1)=j-1; //Se empieza en una anterior para interpolar
   // Se leen 3 latitudes para interpolar los datos
   nc_get_vara_double(ncIdPrec,precId,startp,
                                          countp,partPrecVal);
   //Desplazamos el índice de escritura a su espacio, con
   //latitud(utilLatToWrite*newLonSize*TIME),avanzando al día(i)
   newPrecIndex=i+utilLatToWrite*newLonSize*TIME;
   for(k=0; k<LON; k++){
    if(*(oldLon+k) == -9999.0) continue;
    ... //Aquí se interpolarían los datos
```

```
    //copiamos los datos útiles
    despl_total = k+LON; //Nos movemos a la latitud del medio
    *(escrPrecVal+newPrecIndex)=
                    (float)*(partPrecVal+despl_total);
    //Desplazamos el puntero para que todas las posiciones
    //tengan sus días contiguos.
    newPrecIndex = newPrecIndex + TIME;
   }
  }
 }
 //El índice por el que se tiene que empezar a insertar
 *writeStartp=writeLatIndex;
 //El número de lats que se tienen que escribir
 *writeCountp=utilLatToWrite;
 //Escribimos en el fichero
 nc_put_vara_float(newNcId,newPrecId,writeStartp,
                                  writeCountp,escrPrecVal);
//Índice por el que se empezará a escribir la proxima iter
 writeLatIndex=utilLatToWrite;
```

Como se puede observar, el código resulta algo complejo, pero consigue reducir la cantidad de memoria tantas veces como trozos se indiquen en la constante del programa. El máximo sería si se dividiera la escritura del *dataset* en tantos trozos como número de latitudes tenga el conjunto de datos. Esto permite ser utilizado con *datasets* mucho más grandes que la anterior versión.

Capítulo 4

Análisis de Resultados

En este capítulo se analizarán y compararán los tiempos obtenidos en las implementaciones con OpenMP y CUDA, comentando las limitaciones de cada implementación, sus ventajas y sus desventajas. Cabe destacar que todos los tiempos se toman sobre el conjunto de datos ya optimizado a través del programa descrito en el Apartado 3.1.2. Si esta optimización, el programa habría tardado días en ejecutar debido la estructura de un fichero netCDF, descrita en el Apartado 2.6. Todas las pruebas fueron ejecutadas en el equipo del DATSI de la UPM apodado como "limonero", equipado con un procesador Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz y una tarjeta gráfica Nvidia RTX 2080 Ti. Todas las pruebas se ejecutaron desde un disco duro magnético(HDD).

4.1. Tiempos con OpenMP

En esta sección se analizarán los tiempos obtenidos en cada uno de conjuntos de datos de ejemplo que se han seleccionado y detallando la mejora de rendimiento que se obtiene respecto a la versión inicial(en inglés speed up), representando por tanto cuantas veces más rápido se ejecuta un programa respecto a otro. Se considera la versión inicial del programa la versión secuencial que utiliza *qsort* de la librería estándar de C. La versión que utiliza *qsort* se compiló con gcc utilizando los flags -O3 para la optimización, -lm para incluir la librería *math.h*, -lnetcdf para incluir la librería de netCDF y manipular los datasets y -fopenmp para incluir las librerías de openMp y paralelizar el código. Para la versión que utiliza Thrust en CPU, es necesario utilizar el compilador de Nvidia. Se utilizó nvcc, con los mismo flags anteriormente descritos. Delante del flag -fopenmp es necesario el flag -Xcompiler para pasar el flag directamente a gcc [23].

4.1.1. Ejemplo Pequeño y Mediano

Como se puede ver en la Figura 4.1 la primera implementación con qsort secuencial (1 hilo) tarda cerca de 18 segundos. A medida que aumentamos el número de hilos, el speed up se incrementa. Para obtener un speed up ideal, el speed up debería ser igual al número de hilos. Con 4 hilos, el speed up obtenido es

cercano al ideal, con 3.5 de speed up. Con 8 hilos, el incremento en speed up empieza a bajar, aunque sigue siendo bueno con 6 de speed up. Con 20 hilos el speed up solo aumenta a 9, lo que sugiere un mal aprovechamiento de los hilos.

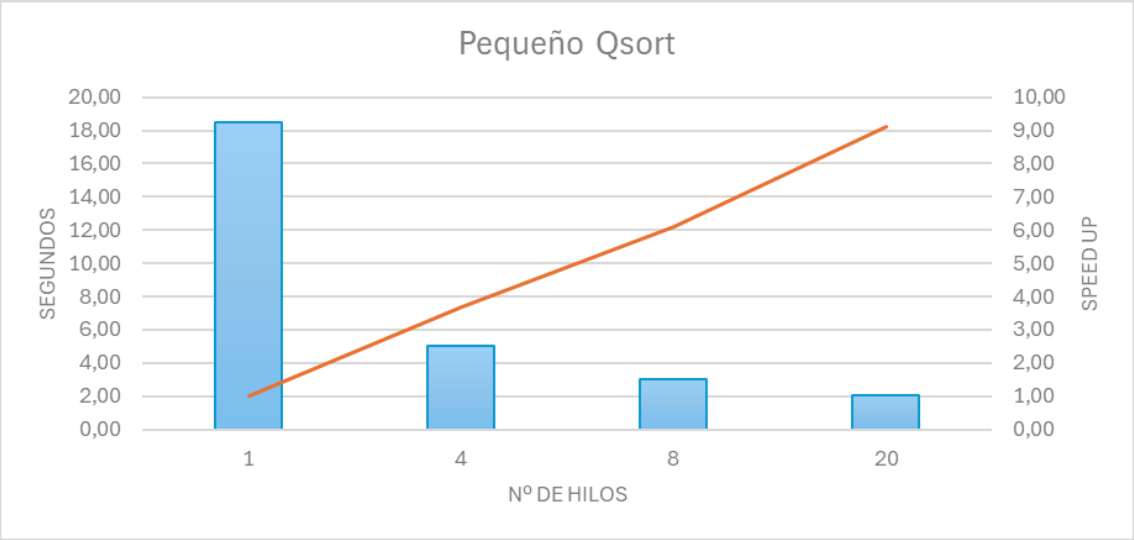

Figura 4.1: Rendimiento qsort en ejemplo pequeño

Con la versión de Thrust, en la Figura 4.2 se observa una tendencia muy similar en el speed up a la implementación en qsort, aunque el tiempo de ejecución se ha reducido a la mitad solo con el cambio de implementación. Esto india que la implementación de Thrust es dos veces más rápida que qsort de la librería de C.

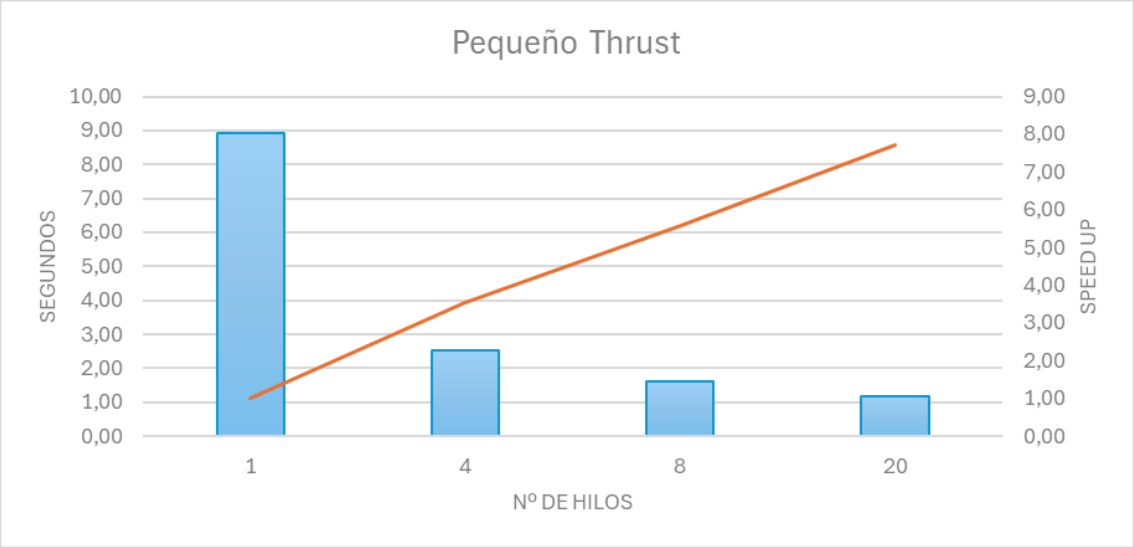

Figura 4.2: Rendimiento Thrust en ejemplo pequeño

En la Figura 4.3 se puede ver la diferencia de tiempos entre ellos, donde Thrust

tarda la mitad cuando se ejecuta secuencialmente, y el speed up de la versión de Thrust representada con la línea verde oscura respecto a la versión inicial con qsort llega a ser de 16 con 20 hilos, y por tanto ejecutando 16 veces más rápido.

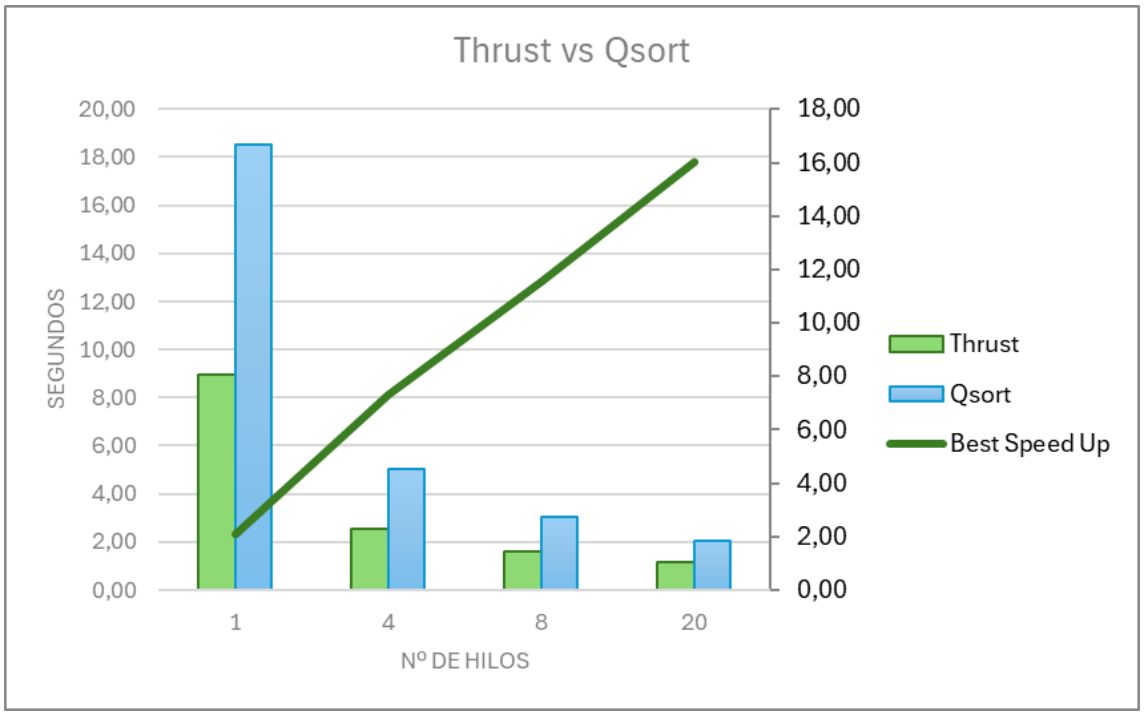

Figura 4.3: Rendimiento Thrust vs qsort en ejemplo pequeño

Con el ejemplo mediano, se pueden ver tendencias similares, con la diferencia del tiempo de ejecución siendo bastante superior. En la Figura 4.4 se puede ver que Thrust secuencial consigue ejecutar cerca de 4 veces más rápido que qsort, pero a medida que aumentamos el número de hilos es diferencia se va reduciendo. En este ejemplo se ve incrementada esa tendencia en la disminución del speed al aumentar el número de hilos.

4.1.2. Ejemplo Grande

En este ejemplo, el tiempo de ejecución del programa aumenta a los minutos. En la Figura 4.5 se puede ver que el speed up se ha reducido considerablemente. Con 4 hilos, el speed up se ha reducido a apenas 1.8 frente al speed up de 3.5 obtenido en los anteriores ejemplos. Con 8 y 20 hilos el speed up no ha mejorado prácticamente lo que indica que hay algo en la implementación evitando que los hilos funcionen correctamente.

En la implementación de Thrust, se observan unos resultados aún peores. En la figura 4.6 se observa que con 4 hilos el speed up es algo mejor que en qsort subiendo a 2.5, pero con 8 y 20 hilos el speed up no ha mejorado, incluso llegando ha empeorar ligeramente con 8 hilos.

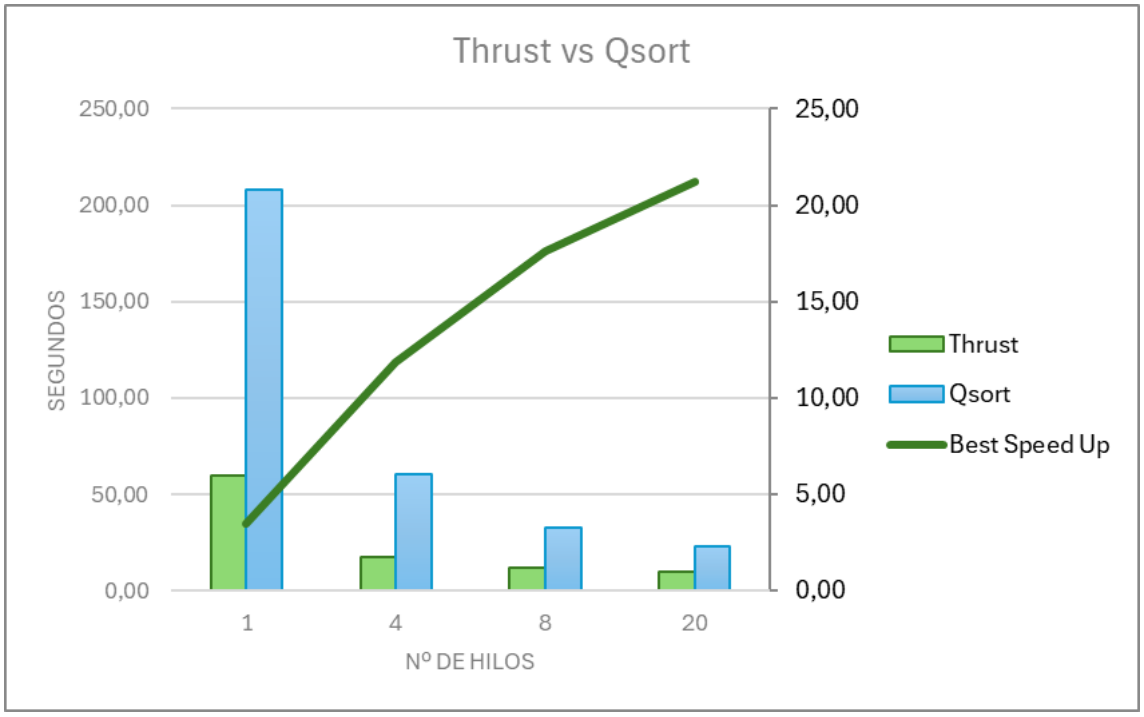

Figura 4.4: Rendimiento Thrust vs qsort en ejemplo mediano

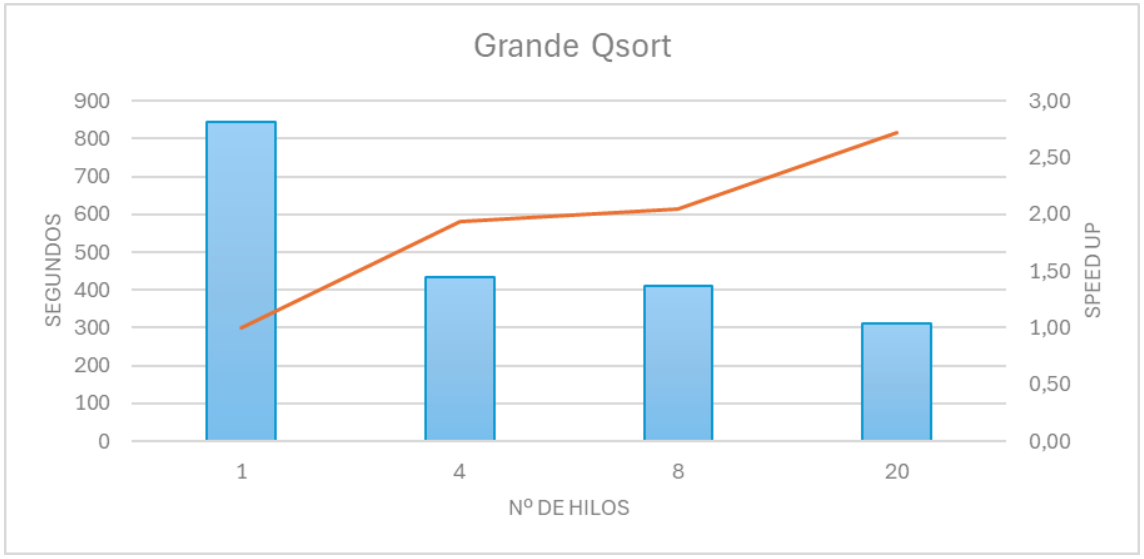

Figura 4.5: Rendimiento qsort en ejemplo grande

Comparando qsort y Thrust en la Figura 4.7, observamos que Thrust sigue comportándose aproximadamente 2 veces mejor que qsort, pero el speed up se ve limitado por el mal funcionamiento de los hilos.

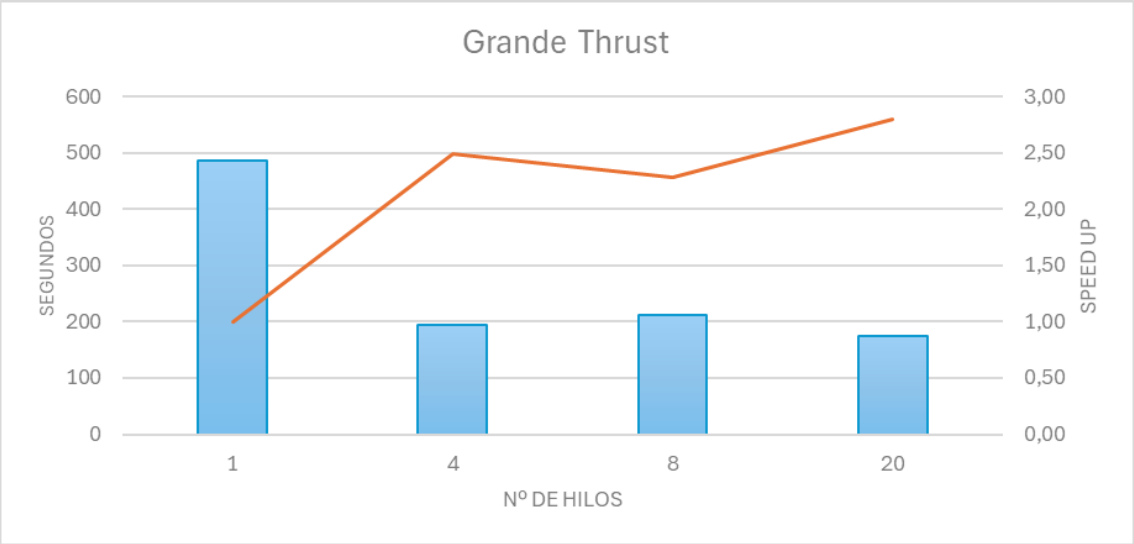

Figura 4.6: Rendimiento Thrust en ejemplo grande

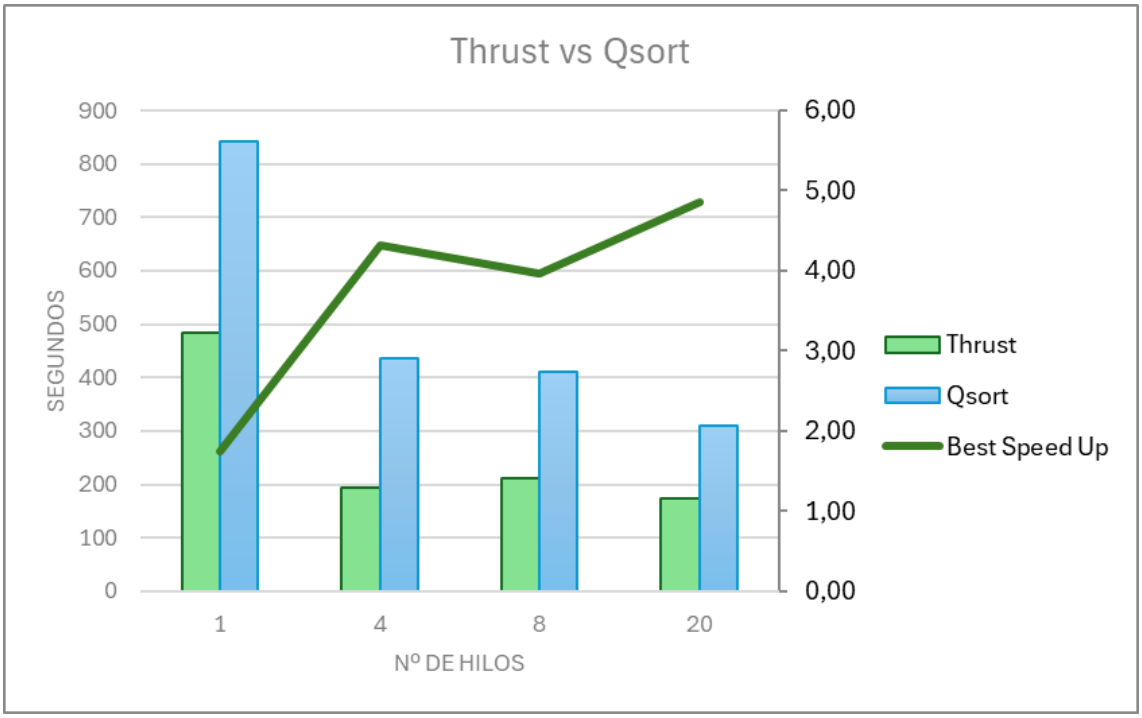

Figura 4.7: Rendimiento Thrust vs qsort en ejemplo grande

4.1.3. Conclusiones

Debido a los resultados obtenidos en el Apartado 4.1.2, que muestran prácticamente ninguna mejoría al utilizar más de 4 hilos, se pensó que podría deberse a la escritura en el nuevo *dataset*, ya que era la única zona que incluía un *#Pragma omp critical* para evitar que dos hilos escribieran al mismo tiempo en el mismo fichero. Para confirmarlo, se decidió utilizar la herramienta de *profiling*

Callgrind que ya se ha utilizado en el Apartado 3.2.2. El *profiling* se realizó al programa implementado con Thrust y paralelizado con 20 hilos. Como se puede ver en la Figura 4.8, se confirman las sospechas anteriormente mencionadas. Se pierde casi un 30 % del programa en el *#Pragma omp critical*. También sorprende que la siguiente función que más consume es el cálculo del *drought code*, en vez del ordenamiento representado por las llamadas a Thrust. Por tanto, una

97.73	5.42	20	main._omp_fn.0
92.92	0.00	19	start_thread
92.92	0.00	19	0x000000000001dac0
27.02	27.02	100 405	GOMP_critical_start
20.27	20.24	100 405	calculateDroughtCode(short*, float*, float*, double*)
11.92	11.92	100 405	void thrust::system::detail::sequential::radix_sort_det...
11.68	11.68	100 405	calculateEstresTermico(short*, short*, short*)
7.45	7.45	100 405	void thrust::system::detail::sequential::radix sort det...

Figura 4.8: Resultados Callgrind en implementación con Thrust y 20 hilos

de las principales limitaciones sería la escritura de datos, y se debe diseñar otra solución para evitar que los hilos se queden bloqueados en el *#Pragma omp critical*. Esta limitación además podría verse reducida si se utilizara un disco duro sólido(SSD) en vez del disco duro magnético utilizado para estas pruebas. Para comprobar como serían los resultados ideales, se realizó una prueba en la que el programa no escribía los resultados, eliminando el *#Pragma omp critical*, pero manteniendo la creación del fichero y las escrituras de los datos básicos del *dataset* como las dimensiones y variables de longitud, latitud y tiempo. Además, estas pruebas se ejecutaron desde un SSD, para comparar la diferencia de rendimiento frente a ejecutar desde un HDD. En la Figura 4.9 se puede ver la comparativa entre la escritura y no escritura utilizando un SSD. Se observa que

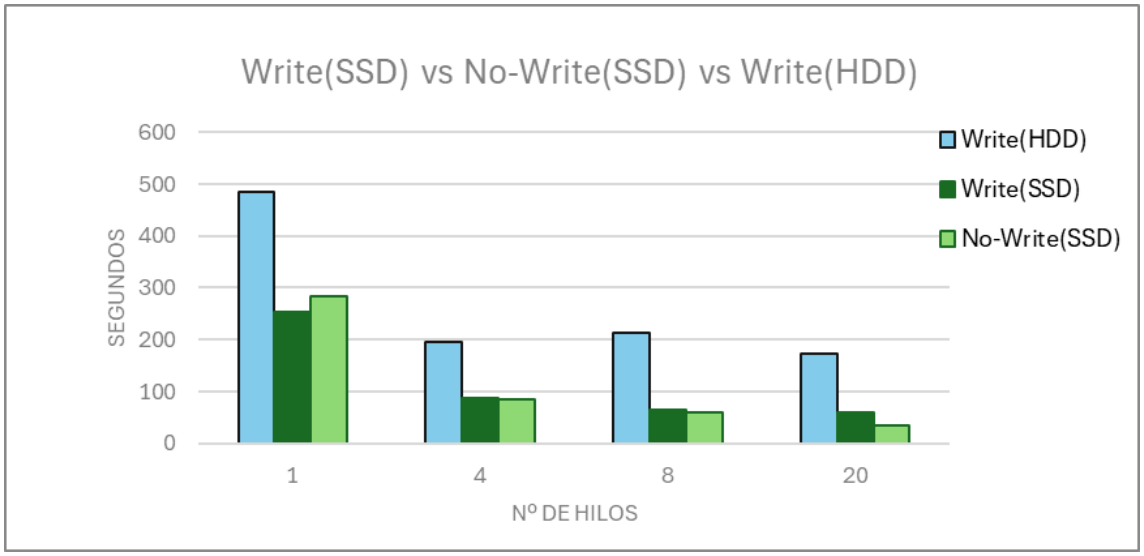

Figura 4.9: Comparación de tiempos frente a la escritura y no escritura en el *dataset*

al ser los tiempos de lectura y escritura mucho más rápidos en un SSD frente a un HDD, los tiempos disminuyen notablemente. Además, se puede ver que con el SSD el problema del *#Pragma omp critical* se reduce ligeramente, permitiendo un mejor aprovechamiento de los 8 hilos. Sin embargo, sigue sin ser suficiente con 20 hilos, como se demuestra en los tiempos conseguidos eliminando la escritura y la zona crítica íntegramente. En el Apartado 5.1 se mencionará una de las posibles soluciones que se han pensado para solucionar este problema.

4.2. Tiempos con OpenMP y CUDA

De la misma forma que en la sección anterior, se realizará un estudio del rendimiento detallando los tiempos y los *speed ups* obtenidos respecto a la versión inicial con el ordenamiento implementado con qsort de la librería estándar de C. Como ya se mencionó en la Sección 3.4, solo se utilizó el ordenamiento en GPU para ordenar la Temperatura, mientras que en todas las versiones se utilizaba Thrust en CPU para ordenar el *drought code*. Tomando en cuenta las conclusiones del apartado anterior, se espera que esta versión sufra también problemas similares cuando se aumenta el número de hilos. Además, como ya se comentó en la Sección 3.4, no se consiguió una implementación que mejorara los tiempos respecto a la versión paralela en CPU, por lo que se espera aún peores resultados.

4.2.1. Ejemplo Pequeño y Mediano

En la Figura 4.10, se puede ver el rendimiento de la implementación *ManyCalls*, que se refiere a la versión que realiza una llamada para el ordenamiento de la temperatura para cada punto, y su versión supuestamente asíncrona. Se observa que ambas implementaciones tienen tiempo similares, por lo que la versión asíncrona y con *streams* no esta funcionando como se esperaba. También destaca que, aunque con 4 hilos los tiempos bajen respecto a la versión secuencial, al aumentar el número de hilos, los tiempos empeoran. Como hipótesis, se piensa que esto se puede deber a una saturación de la tarjeta gráfica que no puede procesar tan rápido como le llegan las llamadas, provocando que los hilos esperen para obtener los resultados.

En la Figura 4.11 se puede ver el rendimiento de la implementación Back-To-Back y de su versión asíncrona descritas en el Apartado 3.4.3. Se observa un mejor aprovechamiento de los hilos, ya que el rendimiento si aumenta al aumentar el número de hilos, con un *speed up* similar al obtenido en la versión paralela. Esto indica que con esta implementación, la GPU si consigue procesar todas las peticiones que le llegan lo suficientemente rápido como para satisfacer a los hilos sin que necesiten esperar. Sin embargo, no se consiguen diferencias entre la versión 'asíncrona' y la síncrona debido a lo explicado en el Apartado 3.4.4.

En la Figura 4.12 se observa que el rendimiento es mucho mejor en la implementación Back-To-Back. Esto indica que Thrust se comporta mucho mejor con

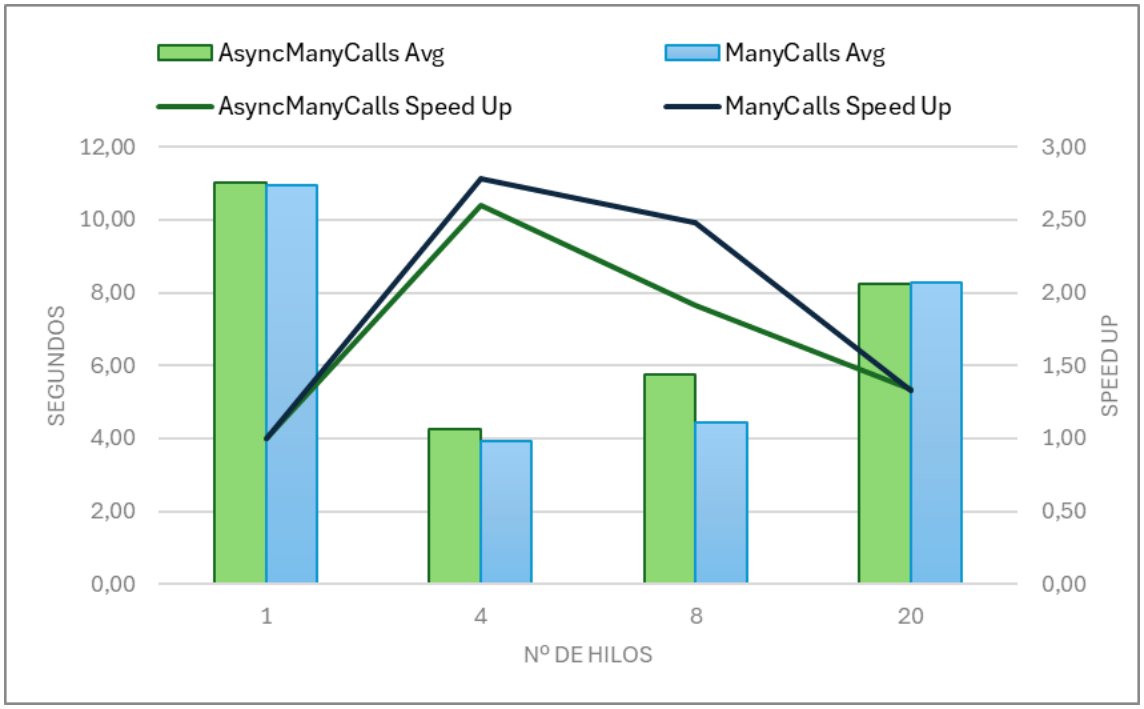

Figura 4.10: Rendimiento ManyCalls y Async ManyCalls en ejemplo pequeño

arrays de grandes dimensiones que con arrays de pequeñas dimensiones. La diferencia de tamaño depende del número de longitudes que tenga el conjunto de datos, y en este caso sería del orden de 200 veces más grande el array que se ordena en la implementación Back-To-Back frente al ordenado en la implementación *ManyCalls*.

En la Figura 4.13 se puede observar las diferencias de tiempos de la primera implementación con qsort, la implementación utilizando Thrust exclusivamente en CPU y la implementación Back-To-Back utilizando GPU. Se observa que en la versión secuencial con GPU mejora ligeramente los tiempos, pero de forma insuficiente para lo que se esperaría al utilizar una tarjeta gráfica. Al aumentar el número de hilos, esta diferencia se va reduciendo hasta ser nula con 20 hilos. Una de las posibles causas es la falsa asincronía de utilizar Thrust con GPU, que provoca que no se aprovecha el paralelismo entre CPU y GPU que podría ayudar a mejorar los tiempos.

En el ejemplo mediano se observan las mismas tendencias, con los métodos asíncronos comportándose de forma similar a los síncronos, y la implementación Back-To-Back obteniendo mejores resultados que la implementación *ManyCalls*. En la Figura 4.14 se observan las mismas tendencias que en el ejemplo pequeño, con la implementación en GPU tardando menos en la ejecución secuencial, y a medida que aumenta el número de hilos siendo igualada a la versión de Thrust en CPU.

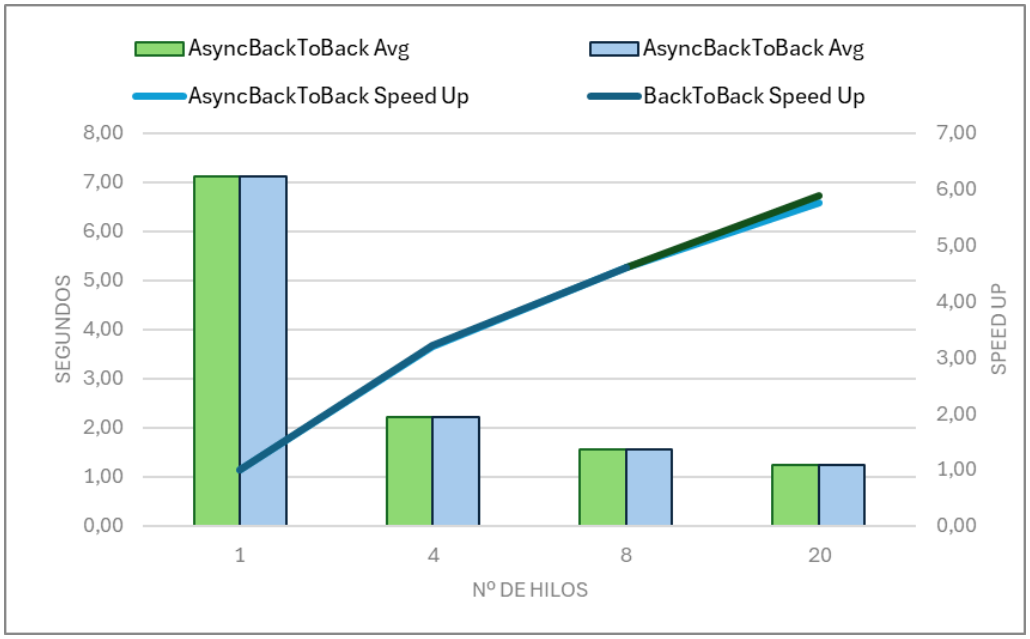

Figura 4.11: Rendimiento Back-To-Back y Async Back-To-Back en ejemplo pequeño

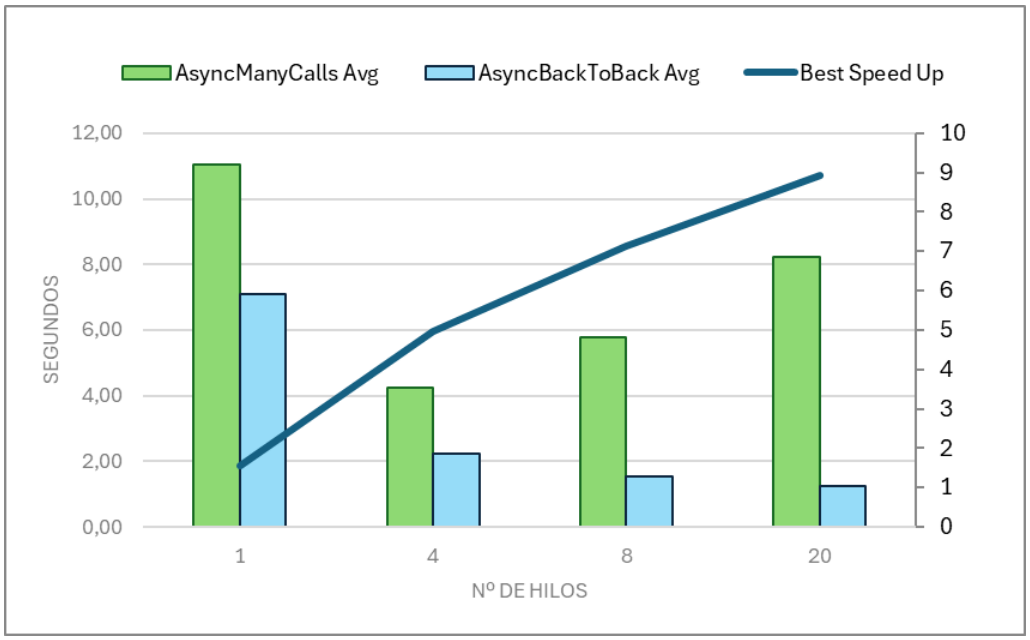

Figura 4.12: Rendimiento Async Back-To-Back vs Async ManyCalls en ejemplo pequeño

4.2.2. Ejemplo Grande

Con el ejemplo grande, se juntan tanto las limitaciones de escritura de la versión paralela, con la saturación y mal rendimiento general de las tarjetas gráficas en la versión de GPU. En la Figura 4.15 se puede observar que el rendimiento no

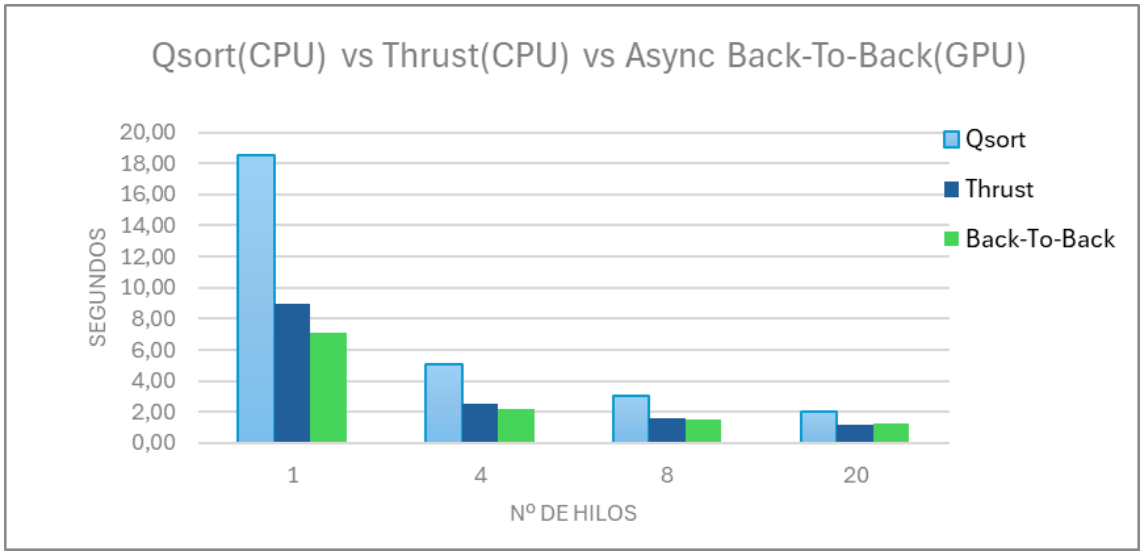

Figura 4.13: Rendimiento qsort vs Thrust vs Async Back-To-Back en ejemplo pequeño

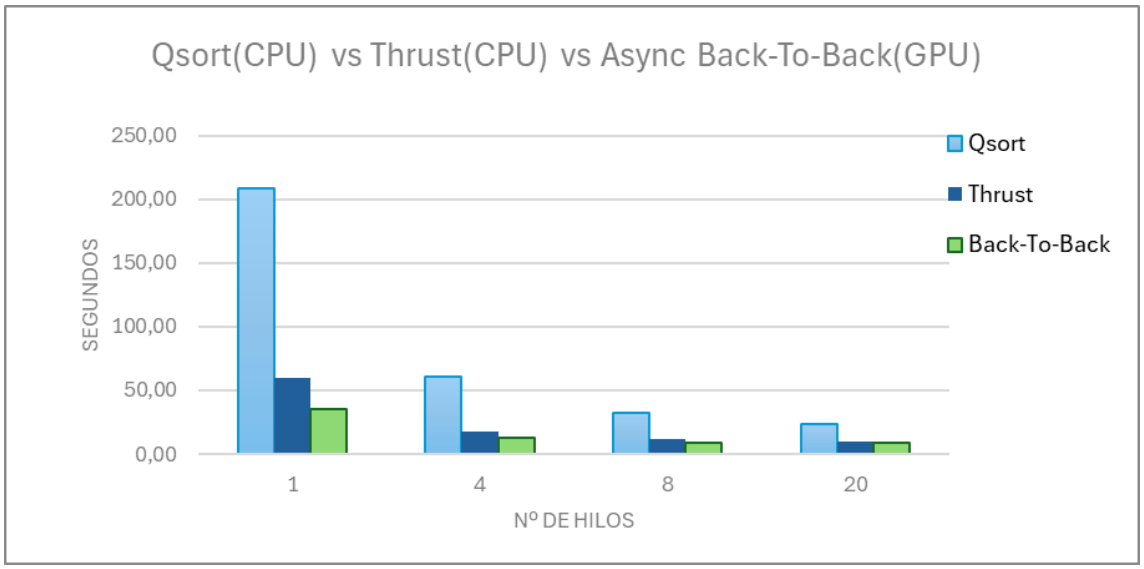

Figura 4.14: Rendimiento qsort vs Thrust vs Async Back-To-Back en ejemplo mediano

varía al aumentar el número de hilos en ninguna de las implementaciones de *ManyCalls*. Se puede ver que la implementación asíncrona tarda algo menos en la versión secuencial, con 4 y con 8 hilos, pero con 20 hilos tarda algo más. Esto se puede deber a una saturación de los *streams* utilizado en la versión asíncrona.

Con la implementación Back-To-Back ocurre algo similar. En la Figura 4.16se puede ver que el rendimiento no mejora al aumentar el número de hilos, si no que empeora. En esta implementación, la versión 'asíncrona' se comporta peor

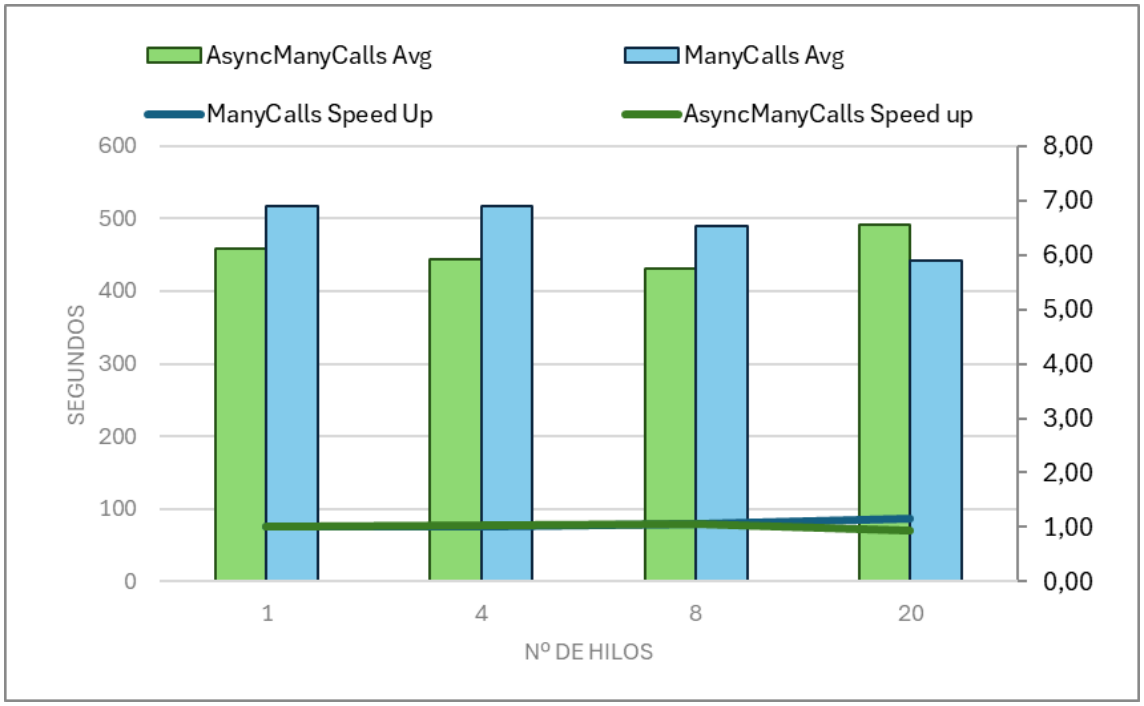

Figura 4.15: Rendimiento de ManyCalls vs Async ManyCalls en ejemplo grande

que la versión síncrona, y al aumentar el número de hilos, el tiempo de ejecución se ve incrementado notablemente. Como en la anterior implementación, esto se puede deber a una saturación de los *streams*. En esta implementación, esta saturación se ve incrementado debido a que el array que tiene que ordenar es cerca de 500 veces más grande que el que se tiene que ordenar en la implementación *ManyCalls* para este ejemplo.

En la figura 4.17 se puede ver que la implementación Back-To-Back ejecuta en un menor tiempo que la versión asíncrona de *ManyCalls*. En cuanto al *speed up* de la versión Back-To-Back frente a la versión inicial secuencial implementada con qsort, solamente consigue ejecutar en la mitad de tiempo, y se mantiene constante al aumentar el número de hilos ya que los tiempos de ejecución son muy similares, como se ha comentado anteriormente.

Por último, en la Figura 4.18 se puede ver la comparativa de rendimiento de la mejor versión de GPU, la implementación Back-To-Back síncrona, frente a Thrust ejecutando en CPU y qsort de la librería estándar de C. Se puede observar que como en los anteriores ejemplo, en la versión secuencial consigue ejecutar ligeramente más rápido al utilizar la GPU frente a Thrust en CPU. Al aumentar el número de hilos, la implementación Back-To-Back mantiene el tiempo de ejecución, llegando a ser superado por la versión inicial con qsort al ejecutar con 20 hilos.

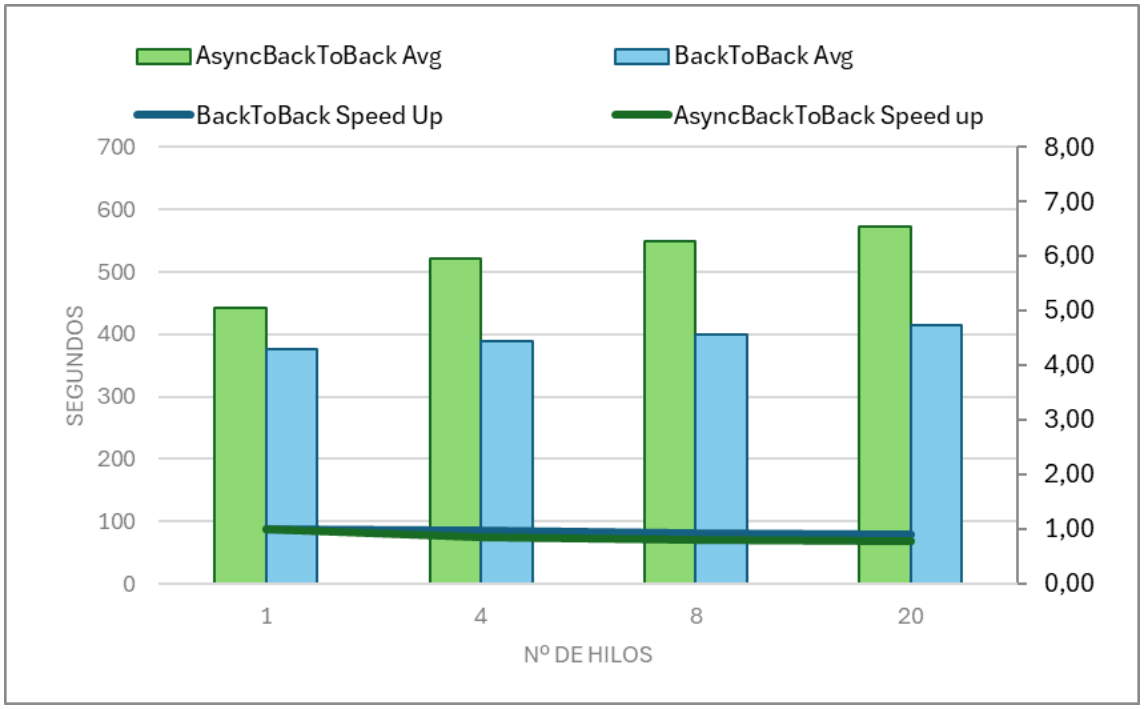

Figura 4.16: Rendimiento de Back-To-Back vs Async Back-To-Back en ejemplo grande

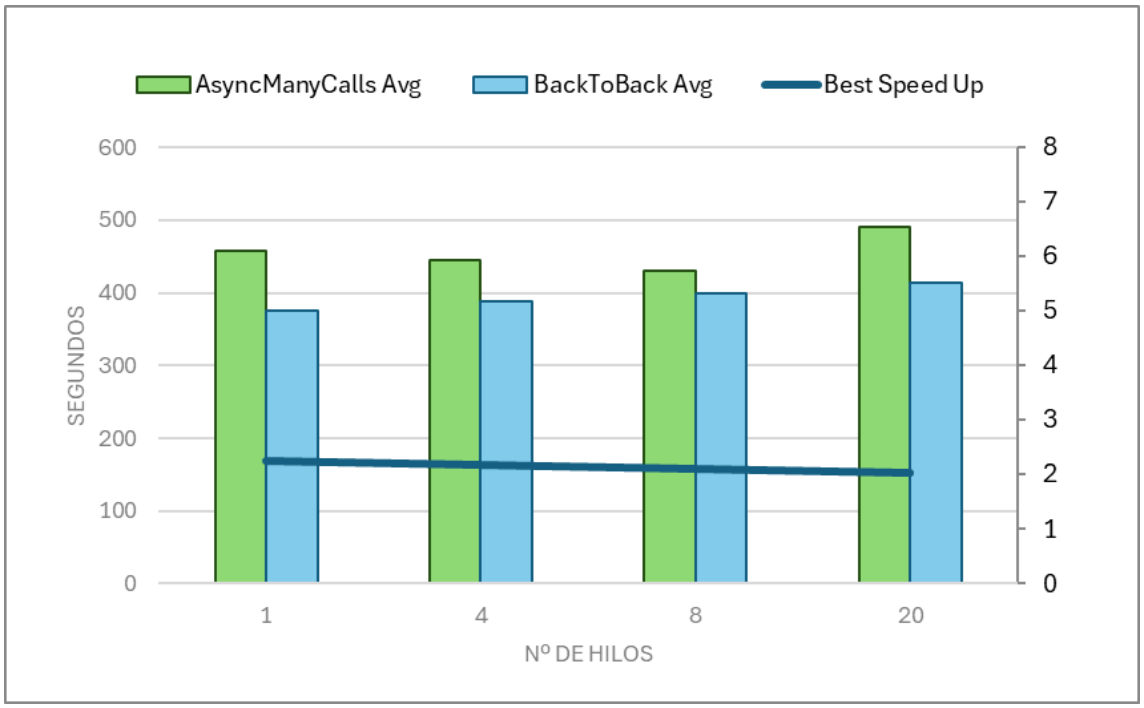

Figura 4.17: Rendimiento de Back-To-Back vs Async ManyCalls en ejemplo grande

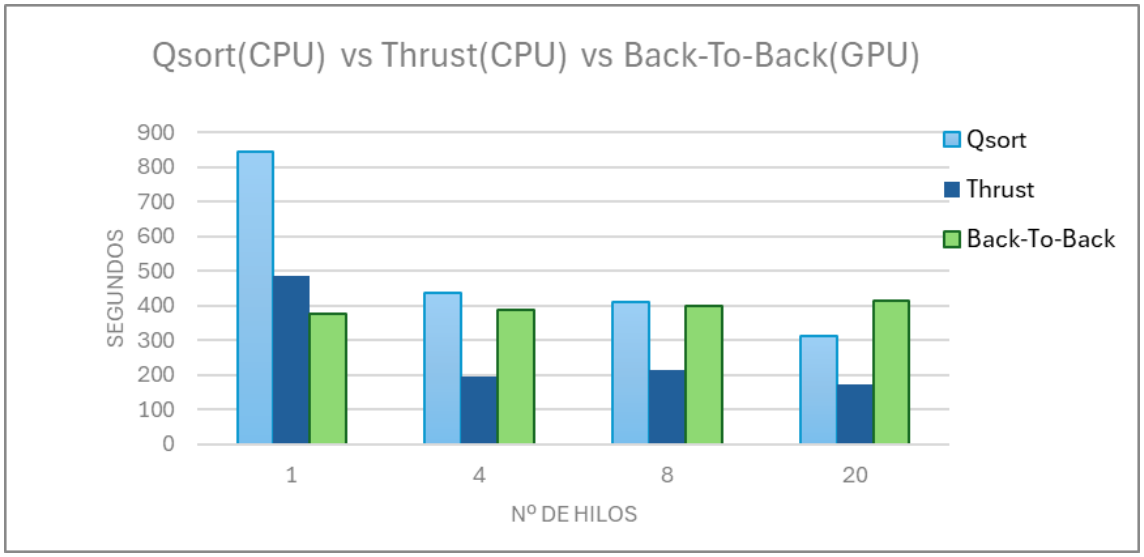

Figura 4.18: Rendimiento qsort vs Thrust vs Back-To-Back en ejemplo grande

4.2.3. Conclusiones

El rendimiento de las implementaciones en GPU ha resultado ser peor que la implementación paralela exclusivamente en CPU utilizando Thrust. En la implementación *ManyCalls*, se piensa que el rendimiento de Thrust en GPU para arrays pequeños no es demasiado bueno, provocando que a pesar de haber conseguido una versión en principio asíncrona, los tiempos de ejecución sean malos ya que los hilos realizan esperas innecesarias. En la implementación Back-To-Back, el rendimiento de Thrust si ha sido bueno, ya que incluso en la versión síncrona se ha conseguido superar ligeramente a la versión en CPU. Sin embargo, al no haber conseguido una versión verdaderamente asíncrona, al aumentar el número de hilos los tiempos de ejecución no disminuyeron tanto como se esperaría.

Capítulo 5

Conclusiones y trabajo futuro

El trabajo realizado en este trabajo de fin de grado permite el cálculo de la anomalía climática en una gran extensión de terreno dividida en extensiones reducidas que permiten un cálculo de alta precisión en apenas minutos. Los programas implementados han sido escritos de forma que no hubiera un límite al tamaño del conjunto de datos, lo que permite su uso para *datasets* incluso más grandes a los usados en este trabajo de fin de grado. La transformación del conjunto de datos para el cálculo óptimo de la anomalía climática permite que las lecturas del conjunto de datos sean 14.000 veces más rápidas, lo que hace posible que el programa de cálculo tarde minutos y se ha aproveche de múltiples hilos en el procesador. Por tanto, se considera que el objetivo principal de este proyecto ha sido completado con éxito.

5.1. Trabajo Futuro

Como Trabajo Futuro sobre los programas realizado en este trabajo de fin de grado, hay varios puntos que se consideran que podrían mejorarse para incrementar el rendimiento. Uno de los principales puntos es la mejora de la implementación con tarjetas gráficas, o para ser más exactos, la búsqueda de una implementación que consiga un paralelismo entre la CPU y la GPU para conseguir un rendimiento máximo. Esto podría conseguirse mediante el uso de *custom_allocators* en Thrust, mencionados en el Apartado 3.4, o con la creación de una nueva función kernel capaz de realizar un ordenamiento óptimo de forma asíncrona.

El segundo punto de mejora es la escritura del nuevo conjunto de datos cuando se ejecutan conjuntos de datos grandes con muchos hilos. Actualmente, como se explicó en Apartado 4.1.3, se pierde un 30 % del tiempo de ejecución esperando en la zona de exclusión mutua necesaria para escribir en el nuevo conjunto de datos. Una de las posible soluciones sería la creación dos buffers, en uno escriben los hilos y cuando está lleno escriben en el segundo y van cambiando. Cada uno lo vuelca al buffer en la zona que corresponda, un solo hilo sería el encargado de escribir el buffer que esta lleno en disco mientras que el resto sigue

escribiendo en el segundo buffer.

Otros de los puntos de mejora es la mejora de la interpolación realizada en el programa de análisis, filtrado y transformación de datos. La interpolación realizada en esta implementación resulta demasiado simple, obligando al programa de cálculo a sustituir algunos valores por la media general. Con una interpolación mejor, esta sustitución sería innecesaria, permitiendo unos cálculos más precisos.

Fuera del alcance de esta implementación, pero apoyado por ella, sería posible debido a la velocidad de ejecución suficiente de la implementación, la creación de suficientes datos como para entrenar una inteligencia artificial capaz de predecir momentos de alto riesgo de intensidad alta de un incendio que permita tomar medidas y potencialmente reducir los daños provocados por un posible incendio.

Capítulo 6

Análisis de impacto

El trabajo realizado en este trabajo de fin de grado permitirá el cálculo de la anomalía climática de una zona extensa con una alta precisión en cuestión de minutos. Esto permite el cálculo de forma rápida de la anomalía climática de una zona concreta con una precisión de metros cuadrados que permitiera la predicción de incendios con alta intensidad. Además, la gran cantidad de datos que es posible generar en apenas minutos permite también su uso para la inteligencia artificial, lo que potencialmente podría ayudar también a la predicción de los incendios. Esto ayudaría a los agentes forestales a tomar medidas que redujeran las consecuencias que pudiera producir un incendio.

La reducción de la intensidad de los incendios es vital actualmente, donde el cambio climática esta aumentando tanto el número como la intensidad de los incendios. Este aumento en intensidad a provocado que en los últimos años se hayan producido algunos de los incendios más devastadores hasta la fecha, no solo en España, sino en todo el mundo [24]. Estos incendios provocan una gran pérdida de diversidad, provocando que se tarden siglos en recuperar estos ecosistemas.

En definitiva, esta implementación que, calculando la anomalía climática, ayuda a la predicción de la severidad de los incendios y permitiendo que se tomen medidas para limitar su severidad, contribuye al Objetivo 15 de los Objetivos de desarrollo sostenible de la Organización de Naciones Unidas [25], contribuyendo a evitar la pérdida de bosques y su biodiversidad.

Bibliografía

[1] C. Europea. «Consecuencias del cambio climático.» (), dirección: https://climate.ec.europa.eu/climate-change/consequences-climate-change_es (visitado 23-05-2023).

[2] F. Chico Zamora. «El Índice de Propagación Potencial (IPP) de Castilla-La Mancha. Herramienta para la predicción del peligro de incendios forestales.» (5 de feb. de 2023), dirección: https://8cfe.congresoforestal.es/es/content/el-indice-de-propagacion-potencial-ipp-de-castilla-la-mancha-herramienta-para-la-prediccion.

[3] S. Swiss Federal Institute for Forest y L. R. WSL. «Fire Weather Indices Wiki | Drought code.» (), dirección: https://wikifire.wsl.ch/tiki-indexd5c6.html?page=Drought+code (visitado 20-04-2024).

[4] J. Alarcón, *Implementación de una aplicación web que permita el cálculo de la evapotranspiración en una determinada zona*, Pendiente de publicación en RUIDERA, 2020.

[5] Almorox.J, *EVAPOTRANSPIRACION. ET. POTENCIAL y ET de REFERENCIA*, Documento utilizado en curso Climatología aplicada a la Ingeniería y Medioambiente de E.T.S de Ingenieros Agrónomos de la UPM. dirección: https://moodle.upm.es/en-abierto/course/view.php?id=16&section=4.

[6] AEMET. dirección: https://www.aemet.es/es/portada.

[7] AEMET. «Datos Diarios de proyecciones climáticas del siglo XXI.» (), dirección: https://www.aemet.es/es/serviciosclimaticos/cambio_climat/datos_diarios?w=2&w2=0 (visitado 21-04-2024).

[8] Copernicus. dirección: https://www.copernicus.eu/es/sobre-copernicus.

[9] UNIDATA, *Network Common Data Form (NetCDF)*, ver. 4.9.2, 21 de abr. de 2024. dirección: https://doi.org/10.5065/D6H70CW6.

[10] S. Ambatipudi y S. Byne. «A Comparison of HDF5, Zarr, and netCDF4 in Performing Common I/O Operations.» (5 de feb. de 2023), dirección: https://arxiv.org/pdf/2207.09503.pdf.

[11] V. Amaral, B. Norberto, M. Goulão et al., «Programming languages for data-Intensive HPC applications: A systematic mapping study,» *Parallel Computing*, vol. 91, pág. 102 584, 2020, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2019.102584. dirección: https://www.sciencedirect.com/science/article/pii/S0167819119301759.

[12] O. ARB, *OpenMP*, ver. 4.5, 15 de oct. de 2021. dirección: https://www.openmp.org/.

[13] T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter y G. Oren, *Quantifying OpenMP: Statistical Insights into Usage and Adoption*, 2023. arXiv: 2308.08002 [cs.DC].

[14] Nvidia, *Thrust: The C++ Parallel Algorithms Library*, ver. 2.1.0, 20 de mar. de 2023. dirección: https://nvidia.github.io/cccl/thrust/.

[15] Unidata, Organization owner of netCDF. dirección: https://www.unidata.ucar.edu/.

[16] R. Rew. «Chunking Data: Why it Matters.» (29 de ene. de 2013), dirección: https://www.unidata.ucar.edu/blogs/developer/en/entry/chunking_data_why_it_matters (visitado 30-05-2024).

[17] Dopico.A, *Programación paralela basada en paso de mensajes (MPI)*, Dept. de Arquitectura y Tecnología de Sistemas Informáticos, Facultad de Informática, Universidad Politécnica de Madrid. dirección: https://datsi.fi.upm.es/~dopico/MPI/MPI.pdf (visitado 30-05-2024).

[18] NetCDF. dirección: https://docs.unidata.ucar.edu/netcdf-c/4.9.2/examples.html.

[19] E. Aspe. dirección: https://github.com/Esteve2604/tfg%5C_anomalia%5C_climatica.

[20] F. e. a. O. S. P. C.Borntraeger, *Valgrind*, ver. 3.18.1, 15 de oct. de 2021. dirección: https://valgrind.org/.

[21] Valgrind, *Cachegrind: a high-precision tracing profiler*. dirección: https://valgrind.org/docs/manual/cg-manual.html.

[22] Valgrind, *Callgrind: a call-graph generating cache and branch prediction profiler*. dirección: https://valgrind.org/docs/manual/cl-manual.html.

[23] Nvidia, *NVIDIA CUDA Compiler Driver NVCC*. dirección: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html (visitado 01-06-2024).

[24] C. C. Garay. «¿Qué consecuencias tienen los grandes incendios forestales en los ecosistemas?» (), dirección: https://www.nationalgeographic.es/medio-ambiente/que-consecuencias-tienen-los-grandes-incendios-forestales-en-los-ecosistemas (visitado 06-03-2024).

[25] ONU, *Objetivos y metas de desarrollo sostenible*. dirección: https://www.un.org/sustainabledevelopment/es/sustainable-development-goals/ (visitado 03-06-2024).