Proyecto de programación orientada al rendimiento.

ARQUITECTURA DE COMPUTADORES 2023

Grupo 80

Grupo 13.

Ignacio Fernández Cañedo	100471955
Belén Gómez Arnaldo	100472037
Luis Gómez-Manzanilla Nieto	100472006
Celia Patricio Ferrer	100471948

ÍNDICE

DISEÑO ORIGINAL

Fluid

Con este componente empieza la ejecución del programa y contiene el *main()*. Primero crea *args* que es una instancia de la clase *ProgArgs* con la que verifica los argumentos de entrada (más adelante explicaremos cómo son los argumentos de entrada). Si los argumentos son válidos, se abre el archivo de entrada para realizar la lectura de las partículas en modo binario y también se abre un archivo de salida para guardar el resultado de las partículas una vez realizada la simulación.

En lo relativo a la simulación del fluido, primero se crea la malla o grid con la función *calcularMalla()* y luego un vector de partículas mediante *leerParticulas()*. Una vez la malla y las partículas están creadas, se llama a *mostrarDatos()* y se inicia la simulación con la función *IniciarSimulacion()*. Las funciones previamente mencionadas se encuentran en *simulación.cpp*. Si la simulación se ha realizado correctamente devuelve 0 y se termina la ejecución.

ProgArgs

Este módulo se encarga de comprobar que los argumentos pasados como parámetros son correctos. Los argumentos son los siguientes:

- *nts:* un número entero positivo que indica en número de pasos de tiempo.
- *Inputfile*: archivo de entrada. Debe ser un archivo existente que debe permitir la lectura. Si no se puede abrir el archivo, se muestra un mensaje de error y termina el programa.
- *Outputfile*: archivo de salida.

Una ejecución por terminal válida es por ejemplo *“./fluid 2000 small.fld small.out”*.

En caso de que el número de parámetros sea incorrecto se terminará la ejecución con el código -1 y mostrando el mensaje "*Error: Invalid number of arguments: n.",* siendo n el número de argumentos introducidos. Al escribir como número de iteraciones cualquier cosa distinta a un número, se muestra el mensaje "*Error: time steps must be numeric.*" y finaliza la ejecución del programa con -1. Adicionalmente, si el número de iteraciones es un número negativo, se imprime el error "*"Error: Invalid number of time steps.*" y se termina el programa con el código de salida -2. Cuando el archivo de entrada no se puede abrir para lectura, se muestra "*Error: Cannot open <file> for reading.*", siendo *file* el archivo de lectura, y se finaliza con -3. Y lo mismo ocurre con el archivo de salida, mostrando “*Error: Cannot open <file> for writing.*”, siendo *file* el archivo de escriturar, y se finaliza con el código -4.

Simulación

Este archivo contiene la ejecución principal para llevar a cabo la completa simulación del fluido. Al inicio, declaramos variables que se van a utilizar a lo largo del archivo. Como estas variables no son constantes

ya que se van a ir actualizando, clang-tidy da error en la declaración, por lo que hemos decidido desactivar dichos errores.

En primer lugar, el archivo contiene la función que calcula las operaciones necesarias para la creación de la malla. Adicionalmente, esta función lee algunas variables del fichero de entrada y realiza cálculos utilizados posteriormente (cuando se muestran los datos por pantalla). Finalmente, se devuelve la malla creada, que explicaremos más adelante.

A continuación, se lee la información de cada partícula del fichero de entrada en binario (*leerParticulas*). Al comenzar, declaramos las variables que se van a leer a continuación. Clang-tidy da error al estar declaradas en una línea, pero creemos que es mejor práctica declararlo todo en una línea para que ocupe menos espacio. La lectura se hace en bloques leyendo todos los valores del *inputfile* relativos a la partícula y se crea con dichos valores. Por último, se cierra el fichero de entrada y se verifica que el número de partículas leído del encabezado del fichero de entrada corresponda con las partículas leídas. En caso de error, se muestra "*Error: Number of particles mismatch. Header: n, Found: m.",* siendo n el número de partículas en leído en el encabezado y m el encontrado tras contar partícula a partícula.

Con respecto a la siguiente función llamada desde *main*, que se llama *mostrarDatos*, como su propio nombre indica, imprime por pantalla los datos relevantes en el mismo formato especificado. Siendo estos, el número de partículas, las partículas por metro, el suavizado, la masa de las partículas, el tamaño de la malla, el número de bloques en total que contiene la malla y el tamaño de los bloques.

La siguiente función definida, *IniciarSimulacion,* consiste en realizar el total de iteraciones pedidas y llevar a cabo la simulación. En primer lugar, se reposiciona todas las partículas de la malla, exceptuando la primera iteración de la simulación. Es decir, que para cada partícula de la malla se inicializan sus valores, se vuelve a calcular su bloque y se cambia de bloque si es necesario.

Posteriormente, se calculan las densidades, las aceleraciones, las colisiones y el movimiento de las partículas. Finalmente, se realiza la función *ResultadosBinarios* que escribe en binario en el *outputfile* las partículas por metro, el número total de partículas y los valores de cada partícula y se cierra el fichero.

Grid

En este archivo se define una clase llamada grid, que representa una malla tridimensional de bloques. En primer lugar, el constructor creará todos los bloques de la malla, que dependerá del número de bloques indicado en el archivo de entrada. Después llamará a *GenerarClaveBloque* que crea una clave identificativa con la posición de cada bloque en la malla. Además, para cada bloque, se calculan y almacenan los bloques adyacentes utilizando el método *calcularBloquesAdyacentes*. El método esValido comprueba si un índice dado de un bloque es válido, es decir, si está en los límites de la malla, que devolverá un valor booleano.

La clase contiene algunas funciones para organizar y operar con las partículas y los bloques como *cambiarBloque*, que dada una partícula evalúa si está en el bloque correspondiente de la malla.

Además, la malla contiene las funciones de *calcularDensidades* y *calcularAceleraciones* que iteran sobre las partículas adyacentes de otra partícula y llaman a funciones de las partículas que cambian sus valores. Decidimos estructurar de esta forma el código para que fuera fácil acceder a las partículas de un bloque y gestionar las partículas adyacentes desde la malla. Además, de esta forma las funciones de la clase partícula sólo cambiarán las propiedades físicas de una partícula, no tienen que hacer comp

Block

Este módulo contiene la creación de la clase Block, la cual representa un bloque dentro de la malla tridimensional. Los bloques se identifican por una key que representa su posición en la malla de la siguiente forma: coordenada i*10000 más la coordenada j*1000 más la coordenada k. Cada bloque contiene dos vectores, uno con las partículas que están actualmente en el bloque y otro con todos los bloques adyacentes a ese bloque.

La clase contiene dos métodos: *addParticles* que añade la partícula al final del vector de partículas y *removeParticles* que elimina la partícula del vector especificada por su id.

Particle

A continuación se define una clase que representa a las partículas de la simulación. Cada partícula se define con un id y coordenadas de su posición, su velocidad y su aceleración que se representan con una estructura de vectores. Además contiene los índices del bloque en el que se encuentra actualmente la partícula y la densidad de esta.

La clase también proporciona otros métodos que se utilizan para calcular o actualizar estas propiedades, como el método Inicializar que asignará a las aceleraciones y a la densidad los valores iniciales necesarios. Para calcular la variación de densidad y de aceleraciones, se utilizarán las funciones *VariacionAceleración* y *VariaciónDensidad*.

Además, la clase Particle proporciona métodos para manejar las colisiones de la partícula(ColisionesEje). Este método recibe 3 parámetros, bmin, bmax y el eje en el que se van a efectuar las colisiones. Clang-tidy da un error en los parámetros, indicando que es fácil confundirlos, pero este error no se elimina cambiando el nombre o el orden de los parámetros por lo que hemos decidido deshabilitar el error, ya que esos son los parámetros que necesita la función. Para mover la partícula se utiliza el método *MoverPartículas*, que actualiza las posiciones y velocidades de la partícula utilizando sus aceleraciones actuales y llamando a la función que calcula las colisiones con los ejes..

Variables globales

Tenemos un archivo llamado *variablesglobales.hpp*, donde se definen las funciones para gestionar las lecturas y escrituras de los archivos binarios y las constantes del programa. Las funciones para leer y escribir archivos binarios utilizan reinterpret_cast que da errores de clang-tidy por lo que hemos decidido desactivar el clan-tidy para estas funciones.

Por otro lado, también se inicializan las constantes de simulación en este archivo para así no tener que volver a inicializarlas al realizar los cálculos y poder acceder a ellas en cualquier momento a lo largo de toda la ejecución del programa. Estas han sido asignadas con el valor especificado en el enunciado.

OPTIMIZACIÓN

Fusión de bucles

En el código final, la función *MoverPartícula* se encarga de todas las operaciones que cambian la posición de una partícula, es decir, las colisiones y el movimiento. En un principio teníamos funciones separadas para las colisiones y para el movimiento de las partículas, lo que hacía que hubiera que repetir 3 veces un bucle que recorría todo el vector de partículas. Hemos unificado estas funciones de forma que solo se realiza un bucle en el que se accede a cada partícula y se realizan todas las operaciones sobre esta en una iteración, mejorando así la localidad temporal.

Claves de la malla

En un principio generábamos las claves de la malla como una concatenación de cadenas de carácteres que hacían referencia a las posiciones de los bloques. Sin embargo, cada vez que queríamos acceder a algún bloque de la malla teníamos que transformar sus coordenadas de enteros a cadenas de caracteres y después concatenarlos, lo que hacía que el programa fuera más lento. Es por eso que decidimos que las claves fueran números enteros formados por las coordenadas de los bloques.

Bloques adyacentes

En la función *calcularDensidades* es necesario recorrer todas las partículas de los bloques adyacentes y del mismo bloque que contiene la partícula para calcular los respectivos valores. Para reducir el número de iteraciones que en un principio se hacían, se ha decidido crear un nuevo atributo en los bloques. Este consiste en un vector que contiene todos los identificadores de los bloques adyacentes al mismo, incluyendo el suyo propio, devueltos por *generarClaveBloque*. De esta forma, cuando se quieran comprobar las partículas adyacentes a otra partícula sólo se comprobarán aquellas que estén en bloques adyacentes al bloque en el que se encuentra la partícula. Esto reduce considerablemente el número de iteraciones y de operaciones que tiene que hacer el programa.

De esta forma, en *calcularDensidades* se iteran los bloques del vector mencionado y las partículas dentro de este bloque. Una vez que se tiene la partícula y una partícula suya adyacente, se procede a realizar la *variacionDensidad* de ambas partículas, calculando las densidades y actualizando su valor como corresponda.

Vector partículas adyacentes

Tanto la función *calcularDensidades* como la función *calcularAceleraciones* hacen operaciones sobre dos partículas adyacentes, es decir, que están separadas a menos de una distancia determinada. Las posiciones de las partículas no cambian en estas funciones. Esto implica que las partículas adyacentes a otra partícula van a ser las mismas cuando se ejecuten estas dos funciones en cada iteración del programa. Es por esto que hemos creado un vector de partículas adyacentes en cada partícula. Cuando se ejecuta la función *calcularDensidades* se almacenan las partículas adyacentes a cada partícula. Después, en *calcularAceleraciones,* se accede directamente a este vector ya que en la función anterior se ha comprobado que eran adyacentes. Esto reduce mucho el tiempo de ejecución del programa porque no hay que comprobar dos veces si una partícula es adyacente con otra.

Cabe destacar que en estos vectores solo se almacenan las parejas de partículas adyacentes una vez, es decir, si dos partículas son adyacentes sólo se encontrará la partícula de mayor identificador en el vector de la partícula con menor identificador porque sino se harían las mismas operaciones sobre cada partícula dos veces.

Colisiones

En un principio teníamos una función de colisiones distinta para cada eje. A nivel de diseño esto no nos parecía lo más óptimo por lo que decidimos crear una única función de colisiones que se puede usar para los tres ejes. Para hacer esto creamos todos los datos de las partículas que tienen 3 dimensiones como estructuras de vectores de 3 dimensiones. Así cuando queremos comprobar las colisiones se llama a la misma función pasando como argumento la dimensión que se está comprobando.

Activar una flag -O3

Se ha activado la flag -O3 para realizar una optimización más agresiva y que se reduzca aún más el tiempo de ejecución. Esta opción es uno de los niveles de optimización más altos con la que se mejora el rendimiento con gran notoriedad. Con esta flag activada, se han visto cambios muy significativos en cuanto al tiempo de ejecución, por lo que se ha visto conveniente hacer uso de ella.

PRUEBAS REALIZADAS

Para la realización de las pruebas funcionales como unitarias, hemos utilizado la herramienta google test.

Unitarias

Las pruebas unitarias se realizan para verificar la funcionalidad de partes individuales del código, como funciones o métodos. En nuestro caso, hemos realizado pruebas unitarias para cada uno de los métodos en las clases *progArgs, block, grid y particle.*

Respecto a las pruebas unitarias de ***progArgs***, una de las pruebas es verificar que el constructor maneja correctamente un número incorrecto de argumentos. Se espera que el programa termine con un mensaje de error específico. Otras pruebas verifican si los pasos de tiempo son un valor no numérico o si son

negativos. Al igual que antes se mostrará un mensaje específico para cada caso. Por último se comprueba si existen los archivos de entrada y de salida y si estos se pueden abrir correctamente.

Las pruebas unitarias de la clase ***block*** consisten en verificar si se añaden o se eliminan correctamente las partículas del bloque, para ello se crea una instancia de *Block* y se llama a *addParticle* y a *removeParticle* comprobando que el tamaño del vector de partículas ha variado respectivamente.

En cuanto a la clase ***grid*** y sus pruebas unitarias, la primera prueba programada es verificar que se crean correctamente todos los bloques en la malla y que cada uno de estos bloques tiene las coordenadas correctas. Otro test es verificar que se añaden correctamente al vector de bloques adyacentes los bloques adyacentes de un bloque. También se verifica que el *block key* se muestra de forma correcta y se ha hecho un test para el método *esValido*, que verifica que da true al introducir un bloque en los límites de la malla y false al introducirlo fuera. Otro test, verifica el método *CalcularBloque*, se añade una partícula a un bloque existente y se verifica que el bloque al que pertenece la partícula es el correcto. Esto se hace también con el test de *CalcularBloqueInicial*.

El método *CambiarBloque* se verifica comprobando que el bloque en el que está una partícula es distinto al bloque en el que está esa partícula al cambiar su posición. Al final se han realizado los test para la función *calcularDensidades*. Este test crea un vector de partículas y se lo manda a *calcularDensidades*, posteriormente se hace una copia del vector y se calcula su densidad. Finalmente se verifica que ambas densidades son iguales. La función *calcularAceleraciones* llama a *VariaciónAceleración* en la clase particle, por lo que el test para verificar esta función se realiza en sus test unitarios.

Por último, también se realizan test de la clase ***particle*** para asegurar que los métodos de la clase funcionan correctamente. Se ha creado un test para verificar que se inicializan correctamente las propiedades de las partículas y otro para comprobar que el cálculo de la distancia entre partículas es correcto. Respecto a las variaciones, se han implementado test para verificar la variación de densidad y de aceleración de las partículas. En este test, se han utilizado números sencillos con lo que poder operar y obtener fácilmente el valor esperado, con el que comparar el resultado tras la llamada de la función. Además, se comprueba que la partícula *particle2* se añade a las adyacentes de *particle2*. En cuanto a las aceleraciones, el test crea dos partículas y establece sus valores iniciales. Después llama a la función *VariacionAceleracion* en *particle1*, pasando *particle2*. Esta función calcula la variación de la aceleración de *particle1* debido a la interacción con *particle2*. Finalmente, el test comprueba si la aceleración de *particle1* es igual a la aceleración negativa de *particle2* para cada eje.

Para los test de las colisiones, primero se prueba la función Colisiones Eje de la clase Particle. Se crea una partícula y establece sus datos iniciales. Se almacena la aceleración inicial de la partícula en *aceleracion_esperada*. Después se llama a la función *ColisionesEje* en particle para cada eje. Esta función debería calcular la nueva aceleración de particle en función de las colisiones con los límites del eje. Finalmente, el test comprueba si la aceleración de particle es igual a *aceleracion_esperada* para cada eje. En cuanto a las segundas colisiones, al igual que antes se crea una partícula y se definen los valores esperados de la posición, la velocidad y la velocidad media después de la colisión. Después se llama a la función *ColisionesEje_2* en particle para cada eje. Finalmente, el test comprueba si la posición, la velocidad y la velocidad media de particle son las esperadas para cada eje. Por ejemplo si particle ha

colisionado con el límite inferior del eje (su posición es menor que *bmin*), se espera que su posición sea bmin y que su velocidad y velocidad media se hayan invertido.

Funcionales

Las pruebas funcionales sirven para verificar que todas las partes del sistema funcionan correctamente juntas. Estas pruebas implican la ejecución de la simulación con diferentes parámetros y la verificación de que los resultados son los esperados. Para ello hemos creado un archivo de pruebas funcionales llamado *ftest*. La estructura de los tests funcionales es la siguiente:

- Se crea una instancia de *ProgArgs* para especificar los parámetros de la simulación, que serán el número de iteraciones, el archivo de entrada y el archivo de salida.
- Se inicia una simulación con estos argumentos. Tras terminar la simulación los resultados están en el archivo de salida escritos en binario.
- A continuación utilizamos la función *leerArchivo* para leer tanto el archivo de salida de nuestra simulación como las trazas proporcionadas. Esta función lee los archivos y almacena el número de partículas, las partículas por metro y la información de cada partícula. La función devuelve una estructura con estos datos, donde la información de cada partícula se guarda en un vector de partículas.
- Por último comprobamos que las salidas de esta función son iguales para el archivo de trazas proporcionado y para el archivo de salida de nuestra simulación. Para ello primero comprobamos que el número de partículas es el mismo, después comprobamos las partículas por metro y después comprobamos que los dos archivos tienen la misma longitud. Por último comprobamos que los valores de los argumentos de las partículas coinciden. Si todos los valores coinciden se pasa el test.

Si alguno de los valores no coincide se muestra un mensaje de error. De esta forma se puede saber qué datos son los que no coinciden ya que los mensajes de error son distintos dependiendo de los datos que no coincidan.

Hemos hecho un test funcional para cada archivo de trazas que se ha proporcionado, es decir, para los archivos large y small de 1 a 5 iteraciones. De esta forma nos aseguramos que funciona correctamente y que se obtienen los resultados esperados.

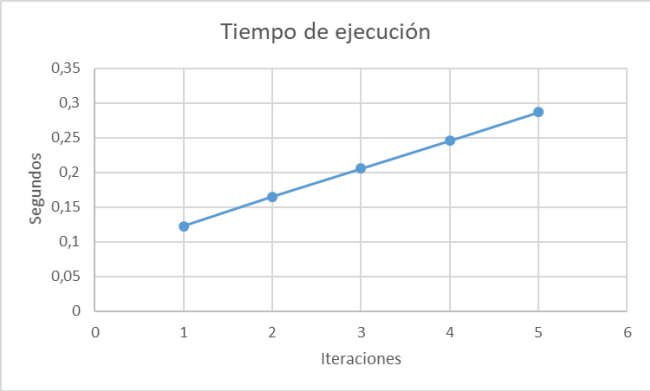
En relación con la complejidad de las pruebas funcionales, clang-tidy emite warnings debido a esto. Hay una alta complejidad porque hemos optado por verificar todos los valores de las partículas. Esto nos permite generar mensajes de error distintos para cada valor. Por este motivo hemos decidido desactivar dichas advertencias.

EVALUACIÓN DEL RENDIMIENTO Y ENERGÍA

En este apartado realizaremos una evaluación del rendimiento del programa, mediremos el tiempo de ejecución y la energía empleada para varias iteraciones. Para ello utilizaremos dos casos: el primero

comparando los resultados para 1, 2, 3, 4 y 5 iteraciones y el segundo comportando los resultados de 100, 200, 400, 600, 800 y 1000 iteraciones.

Empezaremos analizando los tiempos de ejecución del programa:



Tiempos de ejecución caso 1

Tiempos de ejecución caso 2

Como se puede observar, el tiempo de ejecución aumenta de forma lineal con el número de iteraciones. Esto se debe a que el programa tiene una complejidad lineal y al aumentar el número de iteraciones aumenta el tiempo de ejecución proporcionalmente ya que en cada iteración se realiza el mismo número de operaciones.

Consideramos que el rendimiento de nuestro programa es adecuado ya que se ajusta a los valores pedidos. Además, ejecutamos nuestro programa en avignon antes de hacer optimizaciones de diseño (como el vector de partículas adyacentes o las claves de los bloques) y el tiempo de ejecución para 1000 iteraciones era de casi 100 segundos, el doble del tiempo actual.

Además, tras la optimización del vector de partículas adyacentes se redujo el número de instrucciones casi a la mitad, por lo que se mejoró mucho el rendimiento. Esto se debe a que la función *calcularAceleraciones* era de las que más instrucciones tenía y con esta optimización se redujeron.

Ahora haremos el análisis de la energía y la potencia. La potencia se calcula como energía/tiempo. Para ver qué eventos teníamos que analizar usamos el comando per list y nos fijamos en los eventos que empezaban por *power*. Estos eventos son: energy-cores, energy-gpu, energy-pkg y energy-ram. Los resultados obtenidos son los siguientes:

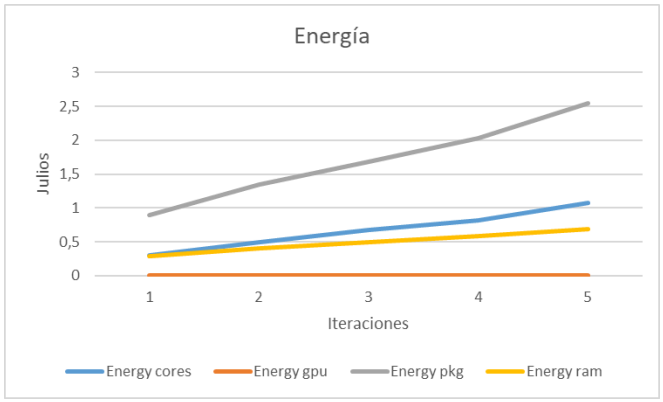


Tabla de consumo de energía caso 1

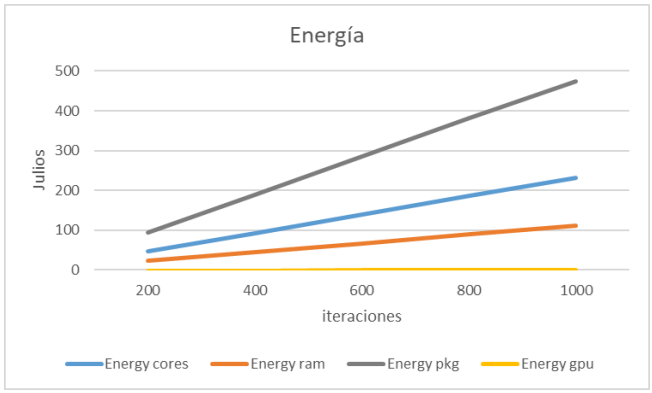


Tabla de consumo de energía caso 2

Como se puede observar, para los dos casos la energía aumenta linealmente con el número de iteraciones. Esto se debe a que un mayor número de iteraciones implica un mayor número de cálculos por lo que implica un mayor consumo de energía.

El evento *Energy pkg* muestra la energía consumida por el paquete de la CPU, que incluye los núcleos de CPU y todos sus subsistemas asociados. Se puede ver que la energía que consume este paquete es mayor que la energía de los núcleos. Esto indica que otros componentes del procesador están consumiendo energía.

También se puede observar que la energía consumida por la RAM es relativamente pequeña en comparación con el paquete y los núcleos. Esto indica que la mayoría de los cálculos se hacen en el procesador y la RAM se utiliza principalmente para almacenar datos y no para calcular operaciones.

Aunque en las gráficas no llega a apreciarse, a partir de las 600 la energía consumida por la gpu es de 0,04- julios mientras que con menos iteraciones es de 0 julios. Esto puede ser porque cuando el programa está utilizando una gran cantidad de memoria, la GPU puede empezar a utilizarse más a medida que la memoria se llena. Esto puede ocurrir cuando se ejecutan muchas iteraciones del programa.

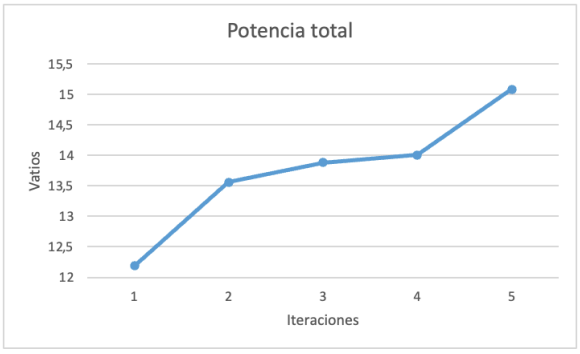


Tabla de la potencia total empleada caso1

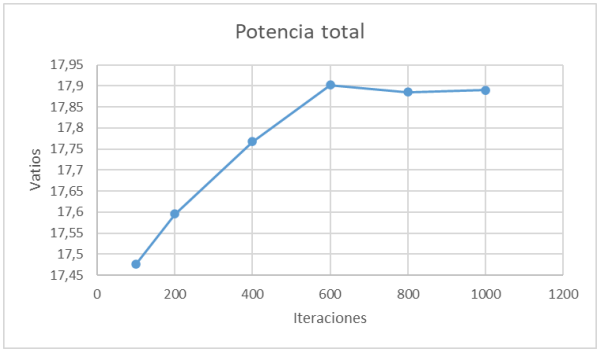


Tabla de la potencia total empleada caso 2

Tras el análisis de la tabla de la potencia total empleada para los diferentes valores de iteraciones, se puede observar como en los primeros cinco valores (de 1 iteración a 5 iteraciones), la potencia consumida aumenta linealmente. Mientras que para el resto de valores de iteraciones, a medida que se aumenta el

número de iteraciones, la potencia consumida también lo hace. Este aumento se realiza de una forma gradual, hasta que a partir de las 600, la potencia empieza a estabilizarse.

Era de esperar que a medida que aumenta el número de iteraciones, la potencia aumenta, ya que son necesarios un mayor número de operaciones. Además, como la potencia consumida está relacionada con el tiempo de ejecución, por lo tanto un menor consumo requiere un tiempo de ejecución más largo, mientras que un tiempo de ejecución menor, emplea un mayor tiempo de ejecución. Pero dado que se ha conseguido alcanzar un tiempo de ejecución razonable, se podría ver que este incremento en la potencia es una compensación aceptable. También se debe tener en cuenta que el consumo de potencia puede estar influenciado por muchos factores, incluyendo el hardware en el que se ejecuta el programa, las optimizaciones del compilador, las condiciones de funcionamiento del sistema, entre otros.

ORGANIZACIÓN DEL TRABAJO

División de tareas

TAREAS	INTEGRANTE	MINUTOS
Creación de la estructura del proyecto	Luis	10
Análisis de argumentos del programa	Ignacio	25
Inicialización de constantes y parámetros de la simulación	Igancio	10
Lectura del archivo de entrada	Ignacio	120
Manejo de error de número de partículas	Igancio	10
Cálculos de inicio de la simulación	Celia	10
Creación de la malla	Belen	50
Creación de vector de bloques adyacentes	Belen	60
Creación de las claves de los bloques y función esValido	Celia	15
Optimización de las claves de los bloques	Celia	20
Mostrar datos de la simulación	Belen	10
Reposicionamiento de partículas	Celia	100
Inicialización de densidades	Luis	5
Cálculo de incremento de densidades	Luis	30
Lectura de las trazas (densinc)	Luis	90
Comprobación de resultados	Luis	10
Corrección de los errores surgidos	Celia	60
Cálculo de transformación de densidades	Celia	10
Lectura de las trazas (denstransf)	Celia	10
Comprobación de resultados	Celia	10

Corrección de los errores surgidos	Igancio	40
Inicialización de aceleraciones	Ignacio	5
Lectura de las trazas (initacc)	Ignacio	10
Comprobación de resultados	Ignacio	10
Cálculo de distancias entre partículas	Luis	30
Cálculo de transferencia de aceleraciones	Belen	120
Lectura de las trazas (acctransf)	Belen	10
Comprobación de resultados	Belen	10
Corrección de los errores surgidos	Ignacio	90
Optimizacion de vector de partículas adyacentes	Luis	90
Primeras colisiones eje x	Celia	40
Primeras colisiones eje y	Igancio	30
Primeras colisiones eje z	Luis	30
Actualizacion de colisiones eje x	Belen	40
Actualizacion de colisiones eje y	Celia	30
Actualizacion de colisiones eje z	Igancio	30
Primer cálculo del movimiento de las partículas	Luis	15
Lectura de las trazas (motion)	Luis	10
Comprobación de resultados	Luis	10
Corrección de los errores surgidos	Ignacio	70
Movimiento de particulas	Celia	35
Vector 3D en las particulas	Luis	40
Escritura del archivo de entrada en binario	Belen	90
Escritura de los archivos CMakeList	Luis	75
Refactorización de la primera mitad del archivo fluid	Ignacio	90
Refactorización de la segunda mitad del archivo fluid	Celia	90
Refactorización del archivo simulación	Belen	70
Refactorización del archivo particle	Celia	60
Refactorización del archivo block	Ignacio	20
Refactorización del archivo grid	Luis	75
Optimizaciones generales	Celia	60
Optimización 1 de la función del movimiento de las partículas	Belen	150
Optimización 2 de la función del movimiento de las partículas	Ignacio	150
Optimización 1 de la función de las colisiones de las partículas	Luis	155
Optimización 2 de la función de las colisiones de las	Celia	120

partículas		
Optimización de fusión de bucles	Belen	45
Creación de los test unitarios de ProgArgs	Celia	40
Creación de los test unitarios de la clase grid	Ignacio	110
Creación de los test unitarios de la clase particle	Belen	95
Creación de los test unitarios de la clase block	Luis	15
Adaptación de los tests de ProgArgs al nuevo código	Celia	30
Adaptación de los tests de grid al nuevo código	Ignacio	40
Adaptación de los tests de particle al nuevo código	Belen	45
Adaptación de los tests de block al nuevo código	Luis	10
Ajuste final de todos los tests	Luis	60
Lectura de los archivos finales	Celia	35
Lectura de los archivos en los tests funcionales	Ignacio	75
Creación de los tests funcionales	Belen	60
Scripts de avignon	Luis	30
Pruebas de simulación en avignon	Ignacio	45
Creación de tablas de tiempo de ejecución	Belen	10
Creación de tablas de energía	Celia	20
Análisis de rendimiento en avignon	Luis	70
Análisis de energía	Celia	70

CONCLUSIONES

Tras la completa realización de esta práctica, con el análisis del rendimiento del programa se ha podido evaluar los diferentes aspectos involucrados en dicho análisis, el tiempo de ejecución total del programa, el consumo de energía y la potencia consumida. Como se ha mencionado ya, el tiempo de ejecución aumenta de manera lineal con el número de iteraciones, lo que se debe a la complejidad lineal del programa. En cuanto al consumo de energía, se observa también un aumento lineal con respecto al número de iteraciones a ejecutar. Y por último, la potencia consumida aumenta linealmente en un principio, volviéndose el aumento más gradual al final, hasta estabilizarse.

Para conseguir un código más óptimo se han aplicado algunas técnicas vistas en clase, como lo es la fusión de bucles. Ya que en un principio, cuando no teníamos el diseño del código bien optimizado ni refactorizado, para realizar diferentes operaciones se utilizaban diferentes bucles. Esto no era nada óptimo ya que dichas operaciones hacían uso de la misma estructura de bucle. Como esto era así, se pudo unificar dichos bucles en uno reduciendo el tiempo de ejecución y mejorar la eficiencia del programa.

Por otro lado, gracias a las clases de laboratorio se ha podido analizar el rendimiento y la energía de nuestro programa. Las clases impartidas sobre estos aspectos han resultado útiles y prácticas ya que se han proporcionado el material y las técnicas necesarias para llevar a cabo la evaluación exhaustiva de la

implementación de esta simulación. Además, estas herramientas han permitido analizar las funciones que eran necesarias optimizar, facilitandonos así en gran medida el trabajo.

Esta práctica nos ha permitido aprender de manera efectiva el uso de diferentes herramientas como lo es clang-tidy, y a analizar y optimizar códigos. Además de conocer más en profundidad el lenguaje de programación C++. La rápida resolución de las dudas por parte de los profesores en el foro general ha sido de gran ayuda para la realización de este proyecto.