Universidad Carlos III

Arquitectura de computadores
Programación Orientada al Rendimiento

Curso 2023-24

Ingeniería Informática, Tercer Curso

Adrián Fernández Galán (NIA: 100472182, e-mail: 100472182@alumnos.uc3m.es)
César López Mantecón (NIA: 100472092, e-mail: 100472092@alumnos.uc3m.es)
Eduardo Alarcón Navarro (NIA: 100472175, e-mail: 100472175@alumnos.uc3m.es)
Paula Subías Serrano (NIA: 100472119, e-mail: 100472119@alumnos.uc3m.es)
Prof. José Daniel García Sánchez, David Exposito Singh, Elías del Pozo Puñal

Grupo de prácticas: 5

Grupo: 81

Índice

1. Introducción al documento

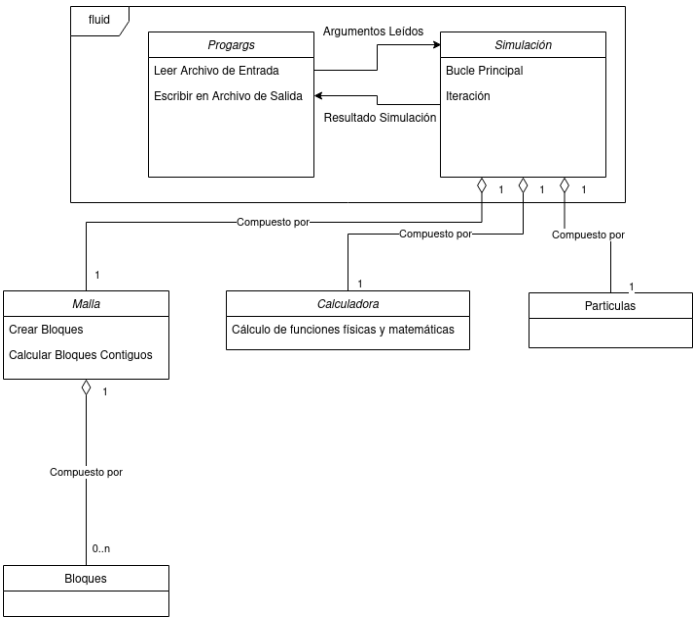
En este documento se recoge el desarrollo de la práctica orientada al rendimiento de la asignatura Arquitectura de Computadores. A continuación se expone el primer diseño del programa, las optimizaciones hechas sobre el mismo, la descripción de las pruebas sobre el código, el análisis del rendimiento (i.e. tiempo de ejecución y consumo energético), la organización del trabajo y las conclusiones derivadas del proyecto. Además, se incluye el conjunto de artículos, documentación y foros consultados que nos han permitido llevar a cabo cada uno de los apartados de este trabajo.

2. Diseño Original

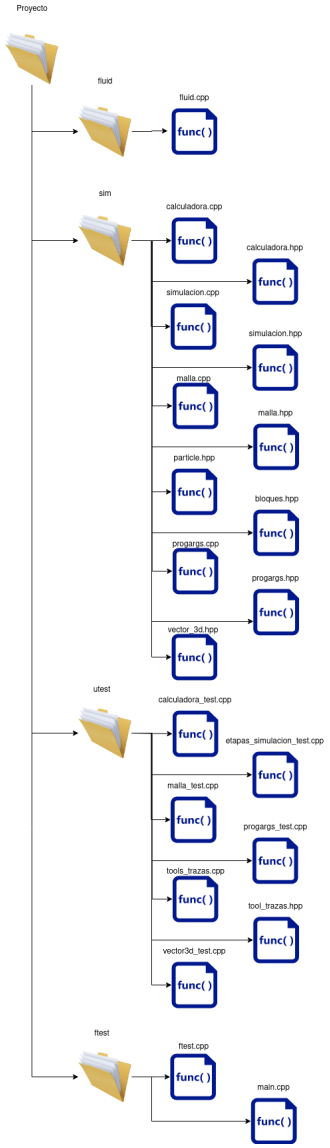
En este apartado se detalla el diseño lógico del programa. A continuación, se detallan las clases implementadas, junto con un breve descripción de lo que representan y las responsabilidades que abarcan. Además, se realiza un breve análisis sobre la decisión de utilizar un *struct de arrays* sobre un *array de structs*; y se evalúa la existencia de goteos de memoria u otros problemas relacionados con la memoria.
El diseño original cuenta con las siguientes clases:

- *Progargs*: implementa la funcionalidad de lectura y escritura en archivos, así como la validación de parámetros de programa.
- *Simulación*: implementa el comportamiento del fluido para un número dado de iteraciones. Para ello contiene una calculadora, una malla, las partículas, el número de iteraciones y el número de partículas.
- *Calculadora*: implementa todas las operaciones matemáticas relacionadas con el modelo. Además, almacena las constantes propias de la simulación.
- *Malla*: contiene los bloques en los que se divide el recinto. Los bloques se guardan en un solo array, y para obtener un bloque en concreto se usa la función *get_pos(i,j,k)*. Cuenta con algunos métodos para crear los bloques, obtener la posición del bloque dentro del array y para obtener los bloques contiguos de un bloque dado.
- *Bloque*: contiene un vector de los índices de las partículas situadas en ese bloque, un vector de los bloques contiguos a él, su posición en el vector de bloques y su posición en forma de coordenadas *i, j* y *k*.
- *Partículas*: struct con los parámetros de cada partícula almacenados en vectores distintos. Más adelante, desarrollaremos por qué decidimos usar esta estructura.

Además, cuenta con una plantilla para la implementación de un vector de 3 valores y sus operaciones.
De esta forma, la funcionalidad del programa queda separada en clases con distintas responsabilidades, permitiendo una codificación más cómoda y favorable al reparto de tareas.
Con esto en mente, nuestro diseño queda representado por el siguiente diagrama:



En cuanto al código, el proyecto está organizado siguiendo la siguiente estructura de ficheros:



**Se han incluído sólo ficheros de código, existen otros directorios con ficheros útiles para los test o shell scripts*

Uso de Struct of Arrays

El diseño de la estructura *Partículas* como *struct of arrays* (SOA) en lugar de *array of estructs* (AOS) se debe a que la primera, dadas las características del programa, es mucho más favorable al uso de la caché. Para explicarlo, nos valdremos de un ejemplo sencillo.
Supongamos que contamos con un computador con un único nivel de caché cuyo tamaño de línea es de 64 bytes y que queremos actualizar la posición de todas las partículas. Usar un AOS supone traerse absolutamente todos los datos de la partícula al acceder a un elemento completo del array (lo que provocará muchísimos más accesos a memoria al necesitar traer más bloques de caché); sin embargo, un SOA nos permite almacenar todos los datos relativos a la posición de las partículas de manera continua en memoria, siendo mucho más favorable al uso de la caché.
Dado que el programa necesita acceder varias veces de manera secuencial a uno de los campos de todas las partículas, determinamos que un SOA presentará un mejor rendimiento que la alternativa.

Goteos de memoria

Después de comprobar el correcto funcionamiento del programa, se ha verificado con la herramienta *valgrind* que no se provocase ningún goteo de memoria en toda la ejecución. También se ha expandido la comprobación mediante el análisis dinámico de accesos a memoria con un perfil extra de compilación que utilice *Address Sanitizer* en la ejecución del programa. Ambas pruebas han resultado exitosas, mostrando la ausencia de goteos en el programa u otros errores relacionados con la memoria.

3. Optimización

Para este apartado se ha medido el tiempo de ejecución del programa original a través de la herramienta *perf*. Obteniendo un tiempo de ejecución de aproximadamente 28 segundos. Si bien es cierto que *perf* otorga una cierta varianza, hemos observado empíricamente que la varianza real puede ser de varios segundos. Esto es importante ya que, en caso de obtener una mejora menor de 1s, no podemos afirmar que efectivamente sea una mejora. Es por esto que para este apartado mediremos el rendimiento de la rutina afectada por la optimización a través de la librería *ctime.*
Esta varianza se explica, seguramente, por procesos de entrada y salida, ya que son las rutinas que presentan más desviación entre ejecuciones.

Punto de partida

La versión original del programa tarda entre 27.777 y 31.168 segundos en ejecutar 1000 iteraciones sobre el archivo *large.fld*. En cuanto a las subrutinas del programa, quedan recogidos los datos en la siguiente tabla:

Rutina	Tiempo (ns)
`lectura`	$2,817 \times 10^6$
`reposicionamiento //Sección 4.3.1`	`886.000`
`colisiones entre partículas //Sección 4.3.2`	$2,4851 \times 10^7$
`colisiones con los límites //Sección 4.3.3`	$1,698 \times 10^6$
`movimiento de partículas //Sección 4.3.4`	`76.000`
`rebote de partículas con los límites //Sección 4.3.5`	`283.000`
`Escritura de salida`	$1,247 \times 10^6$
`main`	$2,77748 \times 10^{10}$

Mediciones realizadas a través de ctime

Uso de *reserve()* para vectores

En el diseño original nos valíamos únicamente del método *push_back()* para introducir nuevos elementos en los vectores de la estructura *Particulas*. De esta forma, hemos observado a través de la herramienta *Vtune* que durante la lectura del archivo se hacían numerosas llamadas al sistema *malloc* y *realloc*. Para reducir el número de estas llamadas, se ha implementado una función *reserve_space()* para la estructura *Particulas* que reserva espacio suficiente para el número de partículas especificado en el archivo de entrada. De esta forma, reducimos el número de llamadas al sistema.
Pese a no mostrar una mejora en tiempo de ejecución (probablemente debido a que la mejora sea baja comparada con la variación debida a detenciones en entrada/salida), sí que observamos una reducción en el número de llamadas al sistema.

Otra forma de iterar por bloques

Para entender esta optimización es necesario analizar en profundidad el funcionamiento de las iteraciones de la colisión de partículas. En las funciones de colisión de partículas se itera para cada bloque, por sus bloques contiguos. Así, evitamos comparar distancias entre partículas de otros bloques.
Una vez dentro de cada par *bloque-bloque contiguo* se compara cada partícula con todas las partículas del bloque contiguo para comprobar si cumplen la condición de colisión, accediendo a los vectores de partículas de ambos bloques, así cómo las posiciones de dichas partículas. Al iterar de esta manera, se realiza *dos veces este acceso a memoria*, al comprobar en ambas direcciones un bloque con sus contiguos. Es decir, se comprueban el par *bloque-bloque contiguo* y el par *bloque contiguo-bloque*. Para evitar duplicar las colisiones, se comparan los índices de partículas permitiendo las colisiones sólo en uno de estos casos. No obstante, eso no evita los dos accesos a memoria.

Para eliminar este doble acceso, se propuso añadir una comprobación extra en el paso de recorrer los bloques contiguos, en el que se comprobaría que el bloque analizado sólo interacciona con bloques con un índice mayor o igual. Así, se descarta el segundo par antes mencionado y se evita traer de memoria los vectores de partículas o las posiciones de este. Este cambio viene acompañado de una modificación de las comparaciones de índice de partícula, para evitar eliminar colisiones por culpa de la doble comparación.
Así, el nuevo bucle elimina accesos a memoria e iteraciones a cambio de 2 comprobaciones extra.
Tras las modificaciones de código, se ha vuelto a medir el rendimiento de la rutina afectada (i.e. *colisiones_partículas()* en 5 ejecuciones, obteniendo una mejora, aunque no es significativa en tiempo de ejecución global.
El tiempo de ejecución medio de ejecución de la rutina queda reflejado en esta tabla:

Antes (ns)	**Después (ns)**
$4,2474 \times 10^7$	$2,0788 \times 10^7$

Tiempo de ejecución de la rutina colisiones_particulas()

Si calculamos la aceleración con estos datos vemos que esta es de aproximadamente 2.

$$S_{colisonesPart} = \frac{4,2474}{2,0788} = 2,0432$$

Siguiendo la Ley de Amdahl, apreciamos que la mejora (aunque sustancial en la rutina) no presenta una reducción muy sustancial en el tiempo de ejecución del programa.

Uso de *-Ofast*

Los compiladores de C++ ofrecen distintas optimizaciones en tiempo de compilación, siendo *-Ofast* un flag que activa las compilaciones más agresivas sobre el código. Estas incluyen todas las optimizaciones *-O3*, además optimizaciones como cálculo rápido de flotantes, entre otras. Estas últimas no cumplen todos los estándares de compilación, por tanto, no sólo mediremos tiempo de ejecución; sino que también debemos validar que seguimos obteniendo un resultado correcto y que superamos las pruebas unitarias para las tolerancias definidas (véase apartado 3, descripción de pruebas).
Al compilar y ejecutar el proyecto usando esta *flag*, vemos una leve reducción del tiempo de ejecución. Además, seguimos superando todas las pruebas, tanto funcionales como unitarias. Por tanto, concluímos que podemos emplear esta opción de compilación y que sí mejora el rendimiento, aunque de manera poco perceptible.

Antes (s)	**Después (ns)**
27,777	26,061

Tiempo de ejecución del programa: al ser una opción de compilación todo el código es afectado.

Conclusiones

Las optimizaciones propuestas no presentan grandes diferencias sobre el tiempo de ejecución total. No obstante, en su conjunto consiguen una reducción de entre 1 y 5 segundos. Ahondaremos más en esto en las pruebas de rendimiento y energía.

4. Descripción de pruebas

Tests Unitarios

Test Progargs

Para probar las clase *progargs*, se realizan pruebas para cada una de las funciones, *constructor, read_head, read_body y write_file*. Para poder probar esta clase es necesario realizar un setup que se encargue de crear ficheros entrada que se almacenan en la carpeta *trz/archivos_utest*. Para probar la función de escritura se utiliza una función auxiliar *compareFile* que comprueba si dos ficheros son iguales.

Test de Calculadora

Para probar la clase Calculadora utilizamos dos instancias de Calculadora, una con 2 partículas y *ppm* igual a 2.0 y la segunda con 4800 partículas y *ppm* igual a 204.0. Para probar cada una de las funciones de Calculadora se utilizan las fórmulas del enunciado para calcular el valor esperado y comparamos con la salida de nuestra implementación. Cabe destacar que para no fiarnos de los redondeos de los double calculamos el error entre el valor esperado y el obtenido y se comprueba que este error sea menor a una tolerancia.
Destacamos los test de las funciones *colision_limite_eje* para cada uno de los ejes, ya que estas funciones tienen 2 flujos de ejecución en base al valor de uno de sus parámetros. Existen test para probar cada una de las posibilidades.

Test de Vector3d

Para probar la clase Vector3d se han realizado tests para cada función, de manera que se prueban con un amplio rango de valores para comprobar el correcto funcionamiento tanto de los operadores sobrecargados como las funciones.

Test De Etapas

Se han desarrollado 2 funciones para el desarrollo de estos tests:

- *CompareSims*: Se encarga de comparar dos instancias de simulación con una tolerancia dada.
- *Load_trz*: Se encarga de volcar los datos de una traza al estado actual de la simulación. Para ello se actualizan los valores de los bloques, ya que almacenan un índice de las partículas que contienen, y de los valores de la clase Simulación.

Para probar las distintas etapas de la simulación cargamos los datos de la traza correspondiente en dos simulaciones, una con los datos de la traza de la etapa anterior y otra con los datos esperados. Esta carga se hace mediante la función *load_trz()*.
Con estos datos, en la primera simulación se ejecuta la etapa a probar y se compara el estado de la simulación con el de la segunda simulación. Se ha definido una tolerancia sobre el error relativo de 1×10^{-12}.

Tests Funcionales

Para la realización de los test ejecutaremos el programa para 0, 1, 2, 3, 4 y 5 iteraciones y comprobamos nuestra la salida con la salida válida proporcionada. Esto se hace de dos maneras: a través de *GoogleTest*, instanciando una simulación con los parámetros adecuados para la ejecución del programa y generando el archivo de salida para luego compararlo mediante la función *compareFile*; a través de un script shell que ejecuta el programa para las iteraciones especificadas y compara la salida a través del comando *diff*. Sin embargo, este último script no compara para 0 iteraciones. Ambas pruebas son equivalentes pese a utilizar métodos distintos.
Para comparar la salida se tienen los archivos de salida en un directorio llamado *out*. La salida se almacena en la ruta *trz/archivos_ftest/*.

5. Pruebas de rendimiento y energía

Para medir el rendimiento del programa se han utilizado las herramientas *perf* (evaluación de tiempo de ejecución y energía) y la biblioteca *ctime* (evaluación de tiempo de ejecución de partes del código).

Tiempo de ejecución

El programa tiene un tiempo de ejecución de 26.211 segundos, con una desviación estándar de un 0.41%. Aunque como ya comentamos en el apartado de Optimizaciones, este último dato no es representativo
Esta medición se ha tomado mediante el siguiente comando:

```
perf stat -r 5 ./build/fluid/fluid 1000 large.fld out.fld
```

El comando nos permite obtener estadísticas medias del rendimiento del programa para 5 iteraciones. Tras su ejecución, obtenemos la siguiente salida:

```
Performance counter stats for 'build/fluid/fluid 1000 large.fld out.fld' (5 runs):

     26.211,34 msec task-clock           #       0,998 CPUs utilized             (+-0,41% )
           131   context-switches        #       0,005 K/sec                     ( +-  9,35% )
             0   cpu-migrations          #       0,000 K/sec                     ( +- 61,24% )
         1.922   page-faults             #       0,073 K/sec                     ( +-  0,03% )
86.046.851.136   cycles                  #       3,283 GHz                       ( +-  0,40% )
210.992.678.706   instructions           #       2,45  insn per cycle            ( +-  0,00% )
38.857.009.336   branches                #    1482,450 M/sec                     ( +-  0,00% )
   248.613.417   branch-misses           #       0,64% of all branches           ( +-  0,13% )

        26,274 +- 0,107 seconds time elapsed  ( +-  0,41% )
```

A continuación se analiza cada una de las estadísticas obtenidas.

- *Task-clock*: tiempo de ejecución. Es interesante observar que hay un uso intensivo de la cpu ya que es un programa con una alta carga aritmética en coma flotante.
- *Context-switches*: número de cambios de contexto que ha sufrido el procesador durante la ejecución del comando. Ha cambiado de tarea un total de 131 veces, un número relativamente bajo.

- *Page-faults*: número de veces que el procesador requería un dato que no estaba en memoria. Es un número bajo que no afecta de manera notable al rendimiento.
- *Cycles*: número de ciclos invertidos en la tarea. Es fácil comprobar la relación entre esta estadística y la primera a través de la frecuencia.

$$\frac{86.046.851.136}{3{,}283\times10^{9}s^{-1}} = 26,21s$$

- *Instructions*: número de instrucciones ejecutadas, junto con el número de instrucciones por ciclo. *IPC = 2.45*.
- *Branches* y *branch misses*: número de bifurcaciones, junto con el desempeño del predictor. El programa tiene un total de 38.857.009.336 bifurcaciones, de las que se predicen incorrectamente un 0.64%. Este buen rendimiento se debe a que la mayoría de bifurcaciones surgen de bucles.

Si comparamos este resultado con la primera versión del código, podemos ver una mejora de rendimiento de 1,5 segundos, lo que se traduce en una aceleración de 1,06.

$$S = \frac{27,77}{26,211} = 1,06$$

Energía

Para obtener estadísticas sobre la energía, empleamos un comando muy similar al anterior.

```
perf stat -r 5 -e ‘power/energy-cores/, power/energy-gpu/, power/energy-pkg/,
        power/energy-ram/’ ./build/fluid/fluid 1000 large.fld out.fld
```

Obtenemos la siguiente salida:

```
Performance counter stats for 'system wide' (5 runs):

       122,5  Joules power/energy-cores/                                 ( +-  1,28% )
         0,02 Joules  power/energy-gpu/                                  ( +- 59,96% )
       256,07 Joules  power/energy-pkg/                                  ( +-  1,18% )
        63,24 Joules  power/energy-ram/                                  ( +-  0,88% )

       26,0617 +- 0,256 seconds time elapsed  ( +-  0,87% )
```

Observamos que el gasto de energía más alto es el indicado por el evento *energy-pkg*, que se corresponde con el consumo energético del paquete de sistema (i.e. cores del procesador, controladores de memoria, caché). En cuanto al resto de medidas, vemos que el mayor consumo viene por parte del núcleo del procesador, seguido de la memoria ram; y, por último, un gasto ínfimo de la tarjeta gráfica. Esto último se explica ya que el programa no emplea la gráfica en ningún momento.

Para saber la energía consumida por nuestro programa debemos sumar las medidas *energy-pkg* y *energy-ram*.

$$E = 256,07 + 63,24 = 319,31J$$

Potencia

La potencia se obtiene como la razón entre la energía y el tiempo:

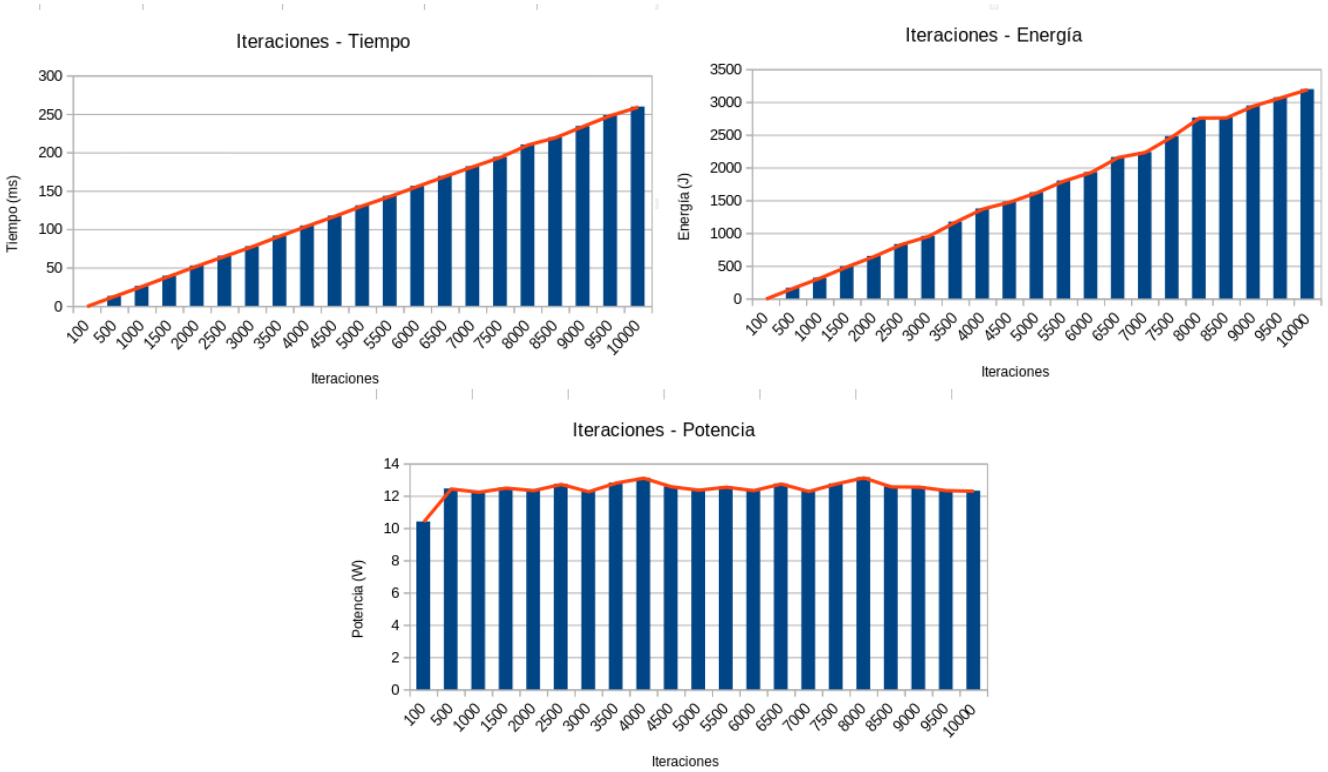
$$P = \frac{E}{T}$$

De las dos medidas anteriores obtenemos que la potencia promedio consumida por nuestro programa para 1000 iteraciones es de:

$$P = \frac{319{,}31}{26{,}211\ s} = 12.182\ W$$

Análisis en función del número de iteraciones

Repitiendo este análisis para distinto número de iteraciones (de 0 a 10000 iteraciones, con un incremento de 500 iteraciones en cada paso) obtenemos las siguientes gráficas:



Tanto la gráfica que representa el tiempo de ejecución contra el número de iteraciones como la gráfica que presenta el consumo de energía contra el número de iteraciones parecen seguir una distribución lineal. Si aplicamos una regresión lineal sobre el conjunto de datos (usamos para esto la herramienta LibreOffice Calc) confirmamos la hipótesis, obteniendo una bondad del ajuste de 0,9998 y 0,9980, respectivamente. Este valor es muy cercano a 1, lo que significa que el modelo de regresión lineal ajusta muy acertadamente los datos.

Array of statistics returned by LINEST function

b_n	b_{n-1}	...	b_1	a	0.02606305	-0.1538729		0.32664562	3.48979039
σ_n	σ_{n-1}	...	σ_1	σ_a	7.1041E-05	0.41525293		0.00336932	19.6945474
r^2	σ_y				0.99985886	0.98311327		0.99798253	46.6269342
F	df				134596.502	19		9398.75496	19
SS_{reg}	SS_{resid}				130089.094	18.3637223		20433560.5	41307.3488

Estructura de la salida de la función LINEST de LibreOffice Calc (izq.) junto con la salida obtenida para las gráficas (derecha)

En cuanto a la gráfica Iteraciones - Potencia, apreciamos que la potencia es relativamente constante, oscilando alrededor de 12,4575 W (valor medio).

Conclusiones

Nuestro programa presenta una complejidad temporal *O(n)* con respecto al número de iteraciones. En cuanto al consumo de energía, crece también de forma lineal. Esto se explica ya que, para un tamaño de archivo constante, los costes en tiempo y energía de las 5 etapas de la simulación son constantes ya que dependen del número de partículas y tamaño de la malla y no del número de iteraciones. Con esto en mente, el bucle principal de función *iterador()* ejecutará las cinco etapas *n* veces; traduciéndose en el coste lineal observado.

$$C = n(C_1 + C_2 + C_3 + C_4 + C_5)$$

Por el contrario, la potencia se mantiene aproximadamente constante, ya que tanto tiempo como energía crecen de forma lineal. Si tratamos la energía y el tiempo de ejecución como funciones lineales, podemos apreciar que la potencia variará menos según aumentemos el número de iteraciones. Es por esto que en la parte derecha del gráfico vemos una menor variabilidad que en la izquierda.

$$\lim_{i\to\infty} \frac{E}{T} = \lim_{i\to\infty} \frac{A_E i + B_E}{A_T i + B_T} = \frac{A_E}{A_T} \equiv cte$$

Nos llama la atención la medida de la potencia para 100 iteraciones, especialmente porque son datos que se alejan mucho de lo esperable tanto en tiempo de ejecución como en consumo energético. Es por esto que hemos repetido las medidas para este número de iteraciones, obteniendo datos completamente distintos y dentro de lo que esperaríamos en un comienzo. Por esto concluímos que la primera medida para 100 iteraciones no debe tomarse en cuenta.

Medidas para 100 iteraciones	
Tiempo de ejecución (s)	2,70145
Energía (J)	32.28
Potencia (W)	11.949

6. Organización del trabajo

Todos los integrantes del grupo han participado de tareas generales como diseño de la interfaz de clases, redacción de la memoria, revisión de errores, labores de optimización, desarrollo de las pruebas de rendimiento y búsqueda de información.

Además del código entregable, se han desarrollado distintos shell scripts y herramientas:

- Shell scripts para la ejecución automática de pruebas de rendimiento.
- Programas en c++ para la traducción a texto en claro de los archivos en binario.
- Un programa en c++ para generar un archivo con dos partículas con la misma estructura que *small.fld* y *large.fld.*
- Hojas de cálculo para recoger y evaluar datos de rendimiento.

Estas tareas han supuesto un trabajo de aproximadamente 15 horas por cada miembro del equipo.

A continuación se adjunta una tabla con la organización del desarrollo de código, así como las horas dedicadas a cada tarea. En esta tabla no quedan reflejadas las tareas mencionadas anteriormente.

Tarea/función	**Persona**	**Descripción**	**Horas**
Lectura y escritura de archivos	César LM	Desarrollo de las funciones *read_file*, *read_head*, *read_body*, *read_till_end*, *write_file* del módulo *propargs.cpp y add_particulas()* del módulo simulacion.cpp.	6h
Movimiento de partículas	César LM	Desarrollo de la función *movimiento_particulas* del módulo *simulacion.cpp*	1h
Desarrollo de los test Calculadora	César LM	Desarrollo de los test unitarios para la clase *Calculadora*	8h
Estructura Vector3d	César LM	Primera versión de la estructura *Vector3d* y templetización de la estructura	3h
Primera y segunda versión de CMakeList.txt	César LM	Solución de errores con cmake y elaboración de la primera y segunda versión plenamente funcionales	4h
Clase Malla malla	Adrián FG	Desarrollo de las funciones *crear_bloques, inicializar_malla, get_pos, bloques_contiguos, existe_bloque*	4h
Bucle principal	Adrián FG	Desarrollo de las funciones *iterador, iteracion*	0,5h
Fase de Población y Reposicionamiento de la iteración	Adrián FG	Desarrollo de las funciones *poblar_malla, reposicionamiento*	2h
Tests Progargs	Adrián FG	Desarrollo de los tests unitarios de la clase *Progargs*	7h
Test Por Etapas	Adrián FG	Desarrollo de los tests unitarios de cada una de las etapas de la clase *Simulacion*	9h
Validación de todos los argumentos del programa	Paula SS	Desarrollo de las funciones de validación *my_is_digit*, *valida_entrada* y *valida_salida.*	3h
Estructura de Progargs	Paula SS	Creación de la clase de progargs, sus atributos y la función *asignar_valores.*	2h
Colisiones entre partículas de la iteración	Paula SS	Desarrollo de las funciones de interacción entre particulas *colisiones_particulas, colisiones_particulas_densidad y colisiones_particulas_aceleracion.*	8h
Verificación de la primera versión del	Paula SS	Comprobación de todas las salidas de las trazas, creación de función *fuera_de_rango* y	10h

programa y corrección de seg fault.		corrección *bloques_contiguos*.	
Tests funcionales	Paula SS	Creación de los tests para probar el funcionamiento del programa completo con distintas iteraciones.	1h
Funciones de cálculos	Eduardo AN	Implementación de las fórmulas operacionales del enunciado a c++, siguiendo las guías de estilo proporcionadas	10h
Etapa 4.3.3	Eduardo AN	Realizada la lógica de la etapa de simulación 4.3.3 - Colisiones de partículas con las paredes	5h
Etapa 4.3.5	Eduardo AN	Realizada la lógica de la etapa de simulación 4.3.5 - Interacciones con los límites del recinto	5h
Estructura Vector3D Op	Eduardo AN	Sobrecarga de los operadores de nuestra estructura Vector3D	2h
Test de Vector3d	Eduardo AN	Creación de los tests para probar el correcto funcionamiento de la estructura Vector3D y que nuestros operadores	3h
Test de Malla	Eduardo AN	Creación de los tests para probar las mallas	1h

7. Conclusiones

Este ha sido, sin duda, el trabajo más complicado al que nos hemos enfrentado en la carrera. La magnitud del mismo, la poca familiaridad con CMake, el lenguaje C++ y los primeros errores en el enunciado han supuesto retos difíciles de superar, sobre todo durante las primeras semanas de trabajo. Además, los errores de herramientas en principio necesarias para la práctica como GoogleTest, CLion o MicrosoftGSL (en el caso de esta última, nos hemos visto obligados a no utilizarla) han supuesto trabas al correcto desarrollo de la práctica. Con todo esto, destacamos como las principales dificultades enfrentadas la organización y división del trabajo y realizar un código idiomático que cumpla con todas las restricciones impuestas por el *clang-tidy* y *clang-format*.

Aun con todo, este proyecto nos ha ayudado a afianzar conocimientos de programación y nos ha introducido al shell scripting y C++. Respecto a la evaluación de rendimiento, el diseño de las pruebas de evaluación de rendimiento nos ha obligado a entender cómo la estructura de código afecta su tiempo de ejecución y consumo energético. También, durante el desarrollo de esta práctica hemos podido consultar una gran cantidad de documentación y debatir con otros grupos de trabajo distintas cuestiones acerca del proyecto, lo que nos ha permitido descubrir nuevos conceptos que, sin duda, aportan a nuestro desarrollo como Ingenieros Informáticos.

8. Documentación consultada

- Funcionamiento de estructuras y funciones nativas de C++: https://en.cppreference.com/w/, https://stackoverflow.com/, https://cplusplus.com/reference
- Uso de *ctime*: https://cplusplus.com/reference/ctime/
- Flags y opticiones de *perf*: https://perf.wiki.kernel.org/index.php/Main_Page
- Opciones de compilación: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- Funcionamiento y opciones de google test: https://google.github.io/googletest/
- Obtención de ejemplos de c++: https://chat.openai.com
- AOS vs SOA:
    - https://www.intel.com/content/www/us/en/developer/articles/technical/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture.html
    - https://stackoverflow.com/questions/17924705/structure-of-arrays-vs-array-of-structures
- Uso de *-Ofast*
    - et. al., E. P. . (2021). Experimental Analysis Of Optimization Flags In Gcc. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, *12*(7), 1875–1879. Retrieved from https://www.turcomat.org/index.php/turkbilmat/article/view/3088
    - D. Branco and P. R. Henriques, "Impact of GCC optimization levels in energy consumption during C/C++ program execution," 2015 IEEE 13th International Scientific Conference on Informatics, Poprad, Slovakia, 2015, pp. 52-56, doi: 10.1109/Informatics.2015.7377807.
- Regresión lineal con LibreOffice Calc: https://wiki.documentfoundation.org/Documentation/Calc_Functions/LINEST