Práctica 1.

Proyecto de programación orientada al rendimiento

uc3m | Universidad Carlos III de Madrid

Arquitectura de computadores

Grupo 80, Equipo 12

Ana María Ortega Mateo: 100472023

Nathalia Moniz Cordova: 100471979

Alberto García de la Torre: 100472039

Índice

Diseño. **3**
- COMPONENTE SIM 3
- COMPONENTE UTEST 5
- COMPONENTE FTEST 5
- OTROS ARCHIVOS 5

Optimización. **6**
Pruebas realizadas. **8**
- GRID_TEST 8
- BLOCK_TEST 8
- PROGARGS_TEST 9
- FUNCIONAL_TEST 9

Evaluación de rendimiento y de energía. **10**
Organización del trabajo. **12**
Conclusiones. **14**

Diseño.

El programa diseñado está organizado en diversos componentes, cada uno con una funcionalidad específica.

- **COMPONENTE FLUID**

La función main() es el punto de entrada del programa y coordina su ejecución, llamando a otros componentes para llevar a cabo la simulación. Su estructura es secuencial, siguiendo los pasos necesarios para la simulación. En primer lugar, establece las bases de la simulación al leer comandos, crear la malla y los bloques. Una vez configurado el punto inicial, procede a realizar la simulación de partículas durante un número específico de pasos.

El código de la función main() se encarga de cargar los argumentos del programa, determinar el número de pasos de la simulación y gestionar la entrada de datos. Luego, inicializa las variables necesarias y realiza la simulación en un bucle que ejecuta pasos cómo añadir partículas, incrementar densidades, transformar densidades, transferir aceleraciones, realizar las acciones de la partícula, eliminar partículas y reiniciar propiedades de las partículas.

- **COMPONENTE SIM**

El componente sim se encarga de realizar toda la simulación del código principal. Está estructurada en distintas partes(ficheros) dependiendo de la función que hemos decidido que desempeñen.

Archivo Progargs

Progargs es el archivo encargado de realizar el tratamiento de argumentos que se reciben por terminal. Por tanto, comprueba que el número de pasos este en formato correcto y sea mayor que cero para poder realizar las iteraciones y que tenga tanto un archivo de lectura para leer las partículas y un archivo de salida para luego poder escribir los resultados finales. Hemos decidido que se encargue de comprobar que los parámetros sean correctos y que tenga el número de argumentos que tiene que tener(3 argumentos).

Al encargarse del manejo de argumentos, hemos decidido que esta componente se encargue de la lectura del fichero pasado como argumento[1], siendo considerados argumentos fundamentales para el main(). Esto es debido a que es necesario la lectura de partículas del fichero de entrada para poder hacer la simulación sobre las partículas correctamente leídas y pasadas a dobles y posteriormente tras la simulación que se copie el resultado en un fichero de salida. Al tener que leer el fichero progargs utilizará dos parámetros fundamentales para la simulación y guardados en una estructura; suavidad y masa, para poder utilizar dichos valores

en el resto del código. Los hemos decidido calcular en progargs ya que se generan a partir de ppm que es un valor que se lee en el fichero de entrada.

Archivo Grid

Grid es el archivo que se encarga de la representación de la malla y en nuestro caso es la encargada de determinar el tamaño de la malla y el tamaño de los bloques que se encuentran dentro de ella. Este último hemos decidido que se realice en grid debido a que la malla está compuesta por bloques, los cuales se encuentran en su interior y dependen del tamaño de la malla. Ambos valores se guardan en una estructura que tiene la malla(nx,ny,nz,sx,sy,sz) que serán imprescindibles para la simulación.

Archivo Block

El archivo Block desempeña un papel crucial en la ejecución de las diversas etapas de la simulación. Esta elección se fundamenta en la disposición de todas las partículas en bloques específicos, lo que significa que en cada iteración de la simulación las partículas residen en un bloque particular. Otro aspecto fundamental es que para realizar ciertas etapas se debía tener en cuenta tanto el bloque en el que está como sus adyacentes y se realizarán los cambios en las partículas en cuestión del bloques en el que se encuentran. Dado que algunas etapas dependen de las anteriores, es necesario registrar los cambios realizados en las partículas de esos bloques durante esa etapa específica. El archivo block cuenta con las siguientes etapas:

1. **Creación de bloques**, para poder realizar la simulación creamos una estructura de bloques, que consta de los números de bloques que tiene la malla(nx*ny*nz).
2. **Anotación de adyacentes**, para saber qué bloques son adyacentes entré sí.

Estas dos fases se ejecutan únicamente una vez, dado que la malla está establecida con un número fijo de bloques, permaneciendo constante a lo largo de toda la simulación. De esta manera, la composición de bloques no varía. En consecuencia, sus bloques adyacentes también son calculados en una única instancia, ya que los bloques mantendrán siempre las mismas relaciones adyacentes. Se incluye a sí mismo en esta lista para facilitar operaciones con las partículas que comparten el mismo bloque, considerándose también adyacentes.

3. **Añadir partículas**, se insertan las partículas dentro de los bloques en función de sus posiciones con los ejes, y el tamaño de los bloques.
4. **Incremento de densidades**, para cada bloque de la malla de simulación, se consideran
5. todos los bloques contiguos así como el bloque actual. Para cada par de partículas i y j, se incrementan las densidades.
6. **Transformación de densidades**, para cada densidad calculada.
7. **Transferencia de la aceleración**, teniendo en cuenta los bloques contiguos del bloque en el que se encuentra la partícula.

8. **Colisiones de las partículas**, únicamente si se encuentran dentro del rango especificado, que justamente coincide con los límites de la malla que se encuentran limitados dentro de un bloque.
9. **Reinicio de las partículas**, para que la densidad y la aceleración vuelvan a sus valores originales.

Fuera del bucle también se encuentra la escritura de los resultados de las iteraciones en el archivo de salida. Esto es debido a que la simulación de las partículas se encuentra en toda esta componente y al ser un requisito propio de la simulación ya al final, se encarga dicha componente de el resultado final de todas las etapas que lo componen guardarlo como resultado de lo que realiza en sí la propia componente. Al principio esta parte se realizaba en progargs ya que se refiere al argv[2], pero decidimos mantenerlo en block al ser un requisito propio de la simulación.

- **COMPONENTE UTEST**

El componente unitest se encarga de realizar las pruebas unitarias de sim.Hay varios archivos para la realización de los test UNITEST. Cada archivo realiza las pruebas necesarias para comprobar el funcionamiento de uno de los archivos .cpp con su archivo sim correspondiente, es decir, progargs_test-cpp se encargará de realizar los test unitarios de programas.cpp.En los test se contemplan tanto los casos en los que el código es correcto, como las situaciones en las que el código debería dar un error(en casos específicos que hemos considerado). Esto nos permite validar exhaustivamente el comportamiento del código y asegurarnos de que se manejan adecuadamente tanto las condiciones normales como las excepcionales.

- **COMPONENTE FTEST**

El componente test se encarga de realizar las pruebas funcionales de toda la aplicación. Para ello, se han cogido las trazas aportadas por aula global y las hemos reorganizado para poder hacer una comparación de lo que se escribía en nuestra simulación con las trazas. SIgue parte por parte todas las funciones que realizamos para la simulación y hacemos una comparación al final simplemente porque si algunos de los procesos falla el resultado final también lo hará.

- **OTROS ARCHIVOS**

Otros archivos que se encuentran en el fichero, son o bien de las trazas o los archivos de lectura(small/large), el archivo final(que podría no estar) y también hemos creado un fichero hpp de constantes globales con todas aquellas constantes que son necesarias a lo largo del proyecto por todas las componentes.

Optimización.

La principal optimización que hemos implementado en nuestro código consiste en la creación de una estructura de bloques adyacentes. Anteriormente, recorríamos todas las partículas y verificábamos una a una si eran adyacentes, lo cual resultaba ineficiente y tenía un impacto significativo en el tiempo de ejecución del programa. Con la introducción de esta optimización, ahora solo recorremos las partículas de los bloques adyacentes.

Esta mejora evita la necesidad de realizar comparaciones individuales de partículas, ya que si dos bloques son adyacentes, sus partículas también lo son. Además, gracias a esta estructura, eliminamos la redundancia de recorrer partícula por partícula en cada ejecución del programa, comparando distancias y evaluando su adyacencia.

Una optimización que se ha tenido en cuenta a la hora de la implementación del código es realizar Arrays fusionados en lugar de arrays paralelos. Hemos ordenado todos los atributos tanto de las partículas y los bloques en estructuras, mejorando la localidad espacial y reduciendo los conflictos que se podrían generar reduciendo la fragmentación de caché.Este enfoque permite un acceso más eficiente a los datos relacionados, ya que reduce la necesidad de acceder a múltiples arrays para obtener información completa sobre una partícula. Nuestras estructuras se componen de las siguientes características:

- **Partículas:** cada partícula contiene los datos necesarios para poder identificarlas dentro de la malla y realizar los cálculos en cada iteración. Sus componentes son: coordenadas de posición x,y,z; coordenadas de vector hv x,y,z; coordenadas de velocidad x,y,z; coordenadas de aceleración x,y,z; densidad de la partícula; id de la partícula; coordenadas x,y,z del bloque en el que se inserta.
- **Bloques:** son el conjunto de bloques que componen la malla y la dividen en secciones más pequeñas. Sus componentes son: id del bloque; array de las partículas que contiene el bloque; array de los bloques adyacentes al actual. Los arrays dentro de la estructura de bloque facilita la accesibilidad de la información de los bloques y acceder tanto a las partículas como a los bloques de manera eficiente y también utilizando id en adyacentes y partículas para directamente acceder al bloque adyacente.
- **ConstantesGlobales:** contiene las constantes smooth (longitud de suavizado) y masa. Se encuentran separadas del resto de constantes porque son calculadas mediante parámetros proporcionados por el archivo de entrada.
- **ConstantesMalla:** aquí se encuentran las constantes relacionadas con la malla. Las constantes nx,ny,nz se obtienen de los parámetros proporcionados por el archivo de entrada; el valor de las constantes sx,sy,sz es dependiente del valor de las constantes

nx,ny,nz.

Gracias a la estructura de las partículas, podemos obtener todos los datos que necesitemos de una partícula en concreto únicamente refiriéndonos a su id, ya que nos referimos a ella utilizándolo y además los actualiza cuando sufren algún cambio. Es decir, si se necesita buscar una partícula en particular dentro de un vector o estructura de datos, el uso de un ID permite evitar búsquedas lineales iterativas. En lugar de recorrer todas las partículas para encontrar la deseada, se puede acceder directamente a ella mediante su ID.

Por otro lado, la estructura de bloques es realmente eficaz cuando queremos comparar dos partículas, ya que tan sólo hay que recorrer los bloques adyacentes (y el bloque propio) de la partícula escogida para identificar cuáles son sus partículas contiguas. De lo contrario, habría que comparar siempre la partícula con todas las demás.

Otra optimización que hemos utilizado es la fusión de bucles en la que en vez de realizar bucles por separado que recorren lo mismo, utilizamos bucles fusionados. Este es el caso de colisiones con partículas, movimientos de partículas y colisiones con los límites. Al recorrer partícula por partícula y no depender de las demás sino de los nuevos datos generados por las demás etapas, realizamos estos tres pasos a la vez por cada partícula y nos ahorramos tener que realizarlo en bucles separados. De esta forma mejoramos la localidad temporal procesando todas esas operaciones para una partícula en un mismo bucle. En vez de tres bucles por separado lo unificamos a uno solo.

También se incluye el intercambio de bucles por accesos secuenciales en vez de con saltos. Los bloques y las partículas se recorren de manera secuencial evitando saltos y mejorando la localidad espacial.

En cuanto a las optimizaciones del compilador, al utilizar el modo release y hacer el build, ejecutando el programa se utiliza la optimización de compilación -O3 la cual ha supuesto un cambio significativo en cuanto al rendimiento y tiempo de ejecución gracias a las distintas optimizaciones que lleva a cabo -03 como la fusión de bucles, que combina varios bucles que realizan tareas similares lo cual reduce el número de instrucciones, o el desenrollamiento de bucles pequeños, ambos reduciendo la sobrecarga de bucles y mejorando la eficiencia del código. Además de esto, -O3 se encarga de reorganizar instrucciones y de optimizar la memoria reorganizando la caché o mejorando el acceso a memoria.

El impacto que ha tenido esta optimización en nuestro programa ha sido bastante notable, inicialmente antes de implementar -O3, utilizando Debug, el tiempo de ejecución del programa para 1000 iteraciones con el archivo “large” es de alrededor de 4,30 minutos. Mientras que aplicando dicha optimización, la ejecución pasó a tardar unos 30 segundos.

Pruebas realizadas.

Con el objetivo fundamental de garantizar el correcto funcionamiento de nuestro programa hemos llevado a cabo un conjunto de pruebas para cada componente esencial de nuestra carpeta de simulación (block, progargs y grid) utilizando la librería de Google Test.

- **GRID_TEST**

Para este primer conjunto de pruebas se ha evaluado primero el que el tamaño de la malla (nx, ny y nz) se calcule de manera correcta. Para ello, hemos establecido el valor del suavizado, ya que dichos parámetros dependen de él, posteriormente construimos el vector de las constantes de la malla llamando a la función **Grid** y finalmente comparamos con los valores correctos esperados.

De igual manera, el tamaño de los bloques (sx, sy y sz) ha sido también comprobado haciendo lo mismo que para el tamaño de la malla, es decir, generamos el vector de las constantes de malla y posteriormente comparamos con los valores esperados.

Asimismo, ha sido necesario evaluar el comportamiento de este componente en casos de error. Con ese propósito, se ha vuelto a generar la malla con sus valores correspondientes y los comparamos con resultados erróneos.

- **BLOCK_TEST**

Hemos realizado pruebas para comprobar todas las etapas que se realizan en el archivo block.cpp. Comprobamos que cada función del archivo realiza correctamente las operaciones que tiene asignadas, y cambia el código de la manera en que está prevista.

En primer lugar, hemos probado la función **crearBloques** que se encarga de crear los bloques y mete sus id en un vector. Para ello se llama a dicha función y se compara el tamaño del vector de bloques, que se corresponde con el número de bloques, con el tamaño de la malla (nx * ny * nz). Esta prueba nos garantiza que la creación de bloques ha sido exitosa, puesto que al comparar con el tamaño de la malla nos aseguramos que todos los bloques caben perfectamente en ésta. Además, para esta misma funcionalidad se ha evaluado su comportamiento en caso de error creando bloques para unos valores de malla y comparando el número de bloques con el tamaño de otra malla.

Del mismo modo, para **anotar_adyacentes** evaluamos que el número de bloques adyacentes de un bloque estén entre 7 y 27 bloques para los valores concretos de nx: 15, ny: 21 y nz 15.

Para las pruebas de error, simplemente para un bloque dado se modificó su número de adyacentes de manera que no se encuentre en el rango adecuado.

En el caso de **añadir_particulas** se llamó a la función añadiendo solo una partícula a un bloque y posteriormente se comprobó que el tamaño del vector de partículas de ese bloque fuese 1 y para su caso de error se agregó una partícula que no pertenecía a los límites de la malla. Por otro lado **ajustar_Lim** fue probada tanto para partículas por debajo del límite inferior como para partículas por encima del límite superior.

Para las funciones que devuelven un valor, comprobamos que lo devuelve correctamente, y que se comporta de la manera esperada en caso de no proporcionar los argumentos esperados. Por otro lado, en el caso de las funciones que realizan los cálculos sobre las partículas como **transferencia_aceleracion**, **movimiento_particulas**, **incrementar_densidades**, **transformar_densidades** y las funciones de **colisiones**, las llamamos con valores concretos y comprobamos que las propiedades de las partículas han cambiado y que sus resultados coinciden con los esperados que han sido calculados manualmente en función estos valores iniciales, así vemos si han modificado el valor esperado y lo han sustituido correctamente.

Cada función tiene por lo menos un test asignado, y cada test realiza un caso específico para poder identificar rápidamente cuál es el fallo en caso de haberlo.

- **PROGARGS_TEST**

Como el archivo progargs.cpp se encarga de la lectura del archivo y la obtención de los parámetros, los tests de esta sección se encargarán de comprobar que se ha leído el archivo correctamente. También, se realizarán varios test encargados de detectar que se lee bien el número de argumentos y pasos proporcionados, además de comprobar que es capaz de abrir el archivo o no en caso de su ausencia. Para estos test también hemos valorado los casos en los que nos daría fallo debido a que tiene salidas de errores y comprobar que en caso de error salten los errores correspondientes.

- **FUNCIONAL_TEST**

Las pruebas funcionales se encargan de verificar la aplicación entera, hemos decidido hacerlo tanto para small como para large utilizando de comparación las trazas. Al haber trazas de hasta cinco iteraciones, se han hecho las comparaciones sobre la cinco ya que si hubiera alguna incongruencia con los resultados la quinta iteración se vería afectada por las anteriores. Las trazas era el único mecanismo que se consideró factible y para evitar hacer los cálculos constantes a mano y comprobar que nuestros resultados daban bien. Cómo estaban organizadas de manera distinta a nuestro archivo de salida, las ordenamos por ids de las

partículas de manera ascendente igual que el archivo de lectura y comparábamos valores. Se ha realizado tanto para small y para large para comprobar que independientemente de cual sea el archivo de entrada los valores son correspondientes y no hay ninguna divergencia.

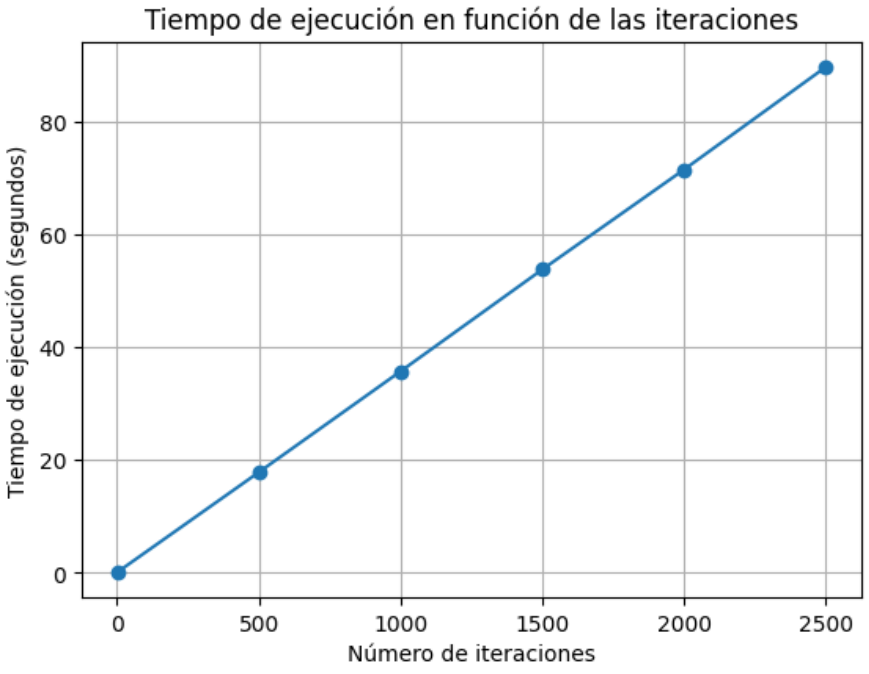
Evaluación de rendimiento y de energía.

Se ha realizado una evaluación tanto del rendimiento como del consumo de energía en función de diferentes números de iteraciones. Este análisis se ha realizado para destacar las variaciones y proporcionar información detallada según la cantidad de iteraciones realizadas.

- Rendimiento.

Para evaluar el rendimiento en términos de tiempo de ejecución hemos incluido la siguiente gráfica que nos muestra el tiempo de ejecución en función del número de iteraciones. Como podemos observar, los tiempos de ejecución aumentan de manera constante (con una media de 17 segundos de diferencia por cada 500 iteraciones) a medida que aumenta el número de iteraciones lo que quiere decir que la complejidad de nuestro código es de carácter lineal, además también sugiere que el coste computacional de nuestro programa en cada iteración también es constante.

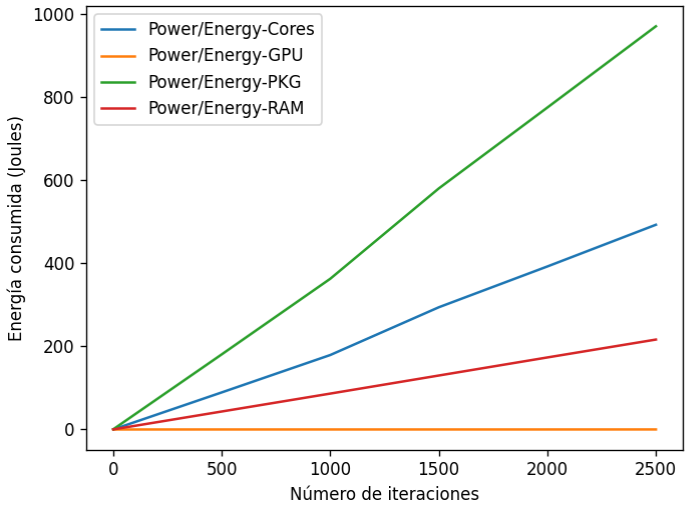
De igual manera, en cuanto a eficiencia, el hecho de que los tiempos de ejecución aumenten de manera lineal con el número de iteraciones es un buen indicador de que nuestro programa tiene una buena capacidad para manejar un mayor número de iteraciones sin que el tiempo de ejecución por iteración varíe significativamente.



- Energía.

Para el análisis de energía hemos añadido la línea **perf stat -a -e** en nuestro script de ejecución para obtener los consumos de para los cuatro eventos diferentes de energía que hemos obtenido al usar **perf list** que son los siguientes:

1. La primera **“Energy-cores”** hace referencia al consumo de energía de los núcleos del procesador el cual va creciendo a medida que se van aumentado el número de iteraciones. Hay una mayor carga de trabajo y actividad en los núcleos durante un período más prolongado de ejecución, realizando mayor número de acciones que aumenta la energía que se necesita para todas las instrucciones que se realizan, cuanto mayor número de iteraciones haya mayor será la energía que se consume. Se especifica más en tareas secuenciales por ello su energía va aumentando, nuestra práctica realiza las instrucciones de manera secuencial.
2. **“Energy-gpu”** hace referencia al consumo de energía de la unidad de procesamiento gráfico. A medida que se incrementan las iteraciones, la GPU se mantiene en cero, esto es debido a que no hay tareas paralelas que se beneficien de su procesamiento y por lo tanto no se activa. No hay ningún tipo de paralelización.
3. **"Energy-PKG"** representa el consumo de energía total del paquete del procesador. Surge un aumento en el consumo de energía total del paquete del procesador a medida que la carga de trabajo, la complejidad de las operaciones y el número en sí de operaciones aumentan. Cuanto mayor número de iteraciones haya crece en mayor tamaño que las demás porque es la que más tareas y componentes del procesador incluye.
4. **“Energy RAM”** representa el consumo de energía asociado a la memoria RAM. Si el número de iteraciones aumenta se producen más accesos a memoria y más cambios en ella tanto de lectura como escritura, por lo que la energía va aumentando cuantas más accesos a memoria haya y por lo tanto si se aumentan las iteraciones se aumentan dichos accesos.



Organización del trabajo.

Al ser fluid el main del trabajo, cada miembro agregaba en él las funciones necesarias con su tarea asignada que se requerían para la simulación. En cuanto a optimizaciones relacionadas con el código cada miembro se encargaba de que su parte estuviese óptima, cumpliese con el clang-tidy y mantuviese relación con las etapas anteriores a dicha tarea,manteniendo así un trabajo secuencial que necesitaba de los unos con los otros para poder proseguir en el trabajo. Son las siguientes:

1. Creación y organización de los archivos del proyecto (Estructura).
2. Configuración de los archivos CMake.

- Análisis de argumentos (Progargs):
    3. Comprobación de los argumentos(función sim_main()).
    4. Definición estructura Particle.
    5. Lectura del archivo de entrada.
    6. Parámetros necesarios de simulación (h y m).
- Malla de simulación (Grid).
    7. Función de crear la malla(calculando valores necesarios nx, ny, nz y sx, xy y sz).
    8. Estructuras que conlleva para posterior aplicación en bloques.
- Etapas simulacion (Block).
    9. Definición estructura bloque.
    10. Crear bloques y anotar adyacentes.
    11. Añadir las partículas en respectivos bloques según su posición.
    12. Incremento de densidades.
    13. Transformación de densidades.
    14. Transferencia de aceleraciones.
    15. Colisiones de las partículas.
    16. Movimiento de partículas.
    17. Colisiones con los límites
    18. Reiniciar valores.
    19. Escritura en el archivo de salida
- Realización de pruebas unitarias:
    20. Realización de pruebas unitarias de progargs.
    21. Realización de pruebas unitarias de grid.
    22. Realización de pruebas unitarias de block(cada uno sus respectivas funciones).

23. Realización de pruebas funcionales.

Integrantes	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Nathalia																							
Alberto																							
Ana																							

Leyenda:

- : menos de 2 horas
- : 2-4 horas
- : más de 4 horas

Los valores mostrados en la tabla son una estimación del tiempo que creemos haber tardado en realizar cada tarea, puesto que no hemos cronometrado el tiempo exacto. Además, en la tabla no se contempla el tiempo invertido en la creación de la memoria ni los arreglos y diversas comprobaciones que hemos hecho del código tras realizar una tarea.

Teniendo todo esto en cuenta, el tiempo medio por persona se encuentra alrededor de las 30 horas.

Conclusiones.

- **Conclusiones de los resultados de la evaluación del rendimiento.**

Habiendo analizado los tiempos de ejecución y el consumo de energía de nuestro programa, hemos llegado a la conclusión de que nuestro programa presenta una eficiencia temporal y energética bastante buena, ya que tanto el consumo de energía como el tiempo de ejecución aumentan ambos de manera lineal a medida que se incrementa el número de iteraciones. En cuanto al uso de recursos, consideramos que se ha hecho un buen uso de los recursos del sistema, puesto que por ejemplo el uso de la GPU es notablemente bajo, además, debido a que el consumo de la RAM es mucho menor que el de la CPU y del paquete de la CPU, sugerimos que nuestro programa hace un uso eficiente de la memoria.

- **Conclusiones del trabajo realizado.**

La práctica ha resultado ser más compleja de lo que parecía, mostrando dificultades y obstáculos que no esperábamos tener en un principio. Para lograr finalizarla, ha sido necesario contar con una organización clara y bien definida del código, siempre teniendo en cuenta qué es lo que están haciendo el resto de los compañeros y escribiendo el código de la manera más legible posible.No sólo hemos entendido en profundidad cómo se trabaja con código c++, sino que además hemos aprendido a sortear los errores mundanos, y hemos encontrado formas de arreglar otros errores más peculiares que han ido apareciendo.

Al tener un código que realizaba distintas tareas separadas que se combinan para obtener un resultado final, el código requería de una estructura bien planificada que no diese paso a ambigüedades en su funcionamiento y a unas pruebas bien definidas. Por ello, la separación del código en distintos archivos, la generación de directorios para cada apartado a realizar, el correcto nombramiento de las funciones y variables, la separación entre declaración y definición, y los test que comprueban el buen funcionamiento del código, han sido cruciales para el control constante de la práctica y su finalización.

El uso del clang-tidy también ha ayudado a tener el código limpio y bien estructurado, además de que ha aportado cambios al código que han optimizado y aligerado el proceso de ejecución. El entorno avignon proporcionado por la universidad nos ha permitido comprobar las distintas optimizaciones realizadas en el código, y así poder comprobar cómo iba poco a poco mejorando con cada cambio.

En conclusión, en cuanto a la relación de este proyecto con el contenido impartido en la asignatura, consideramos que mucho del material dado, en especial las sesiones de laboratorio, nos han sido de gran ayuda tanto para optimizar nuestro programa lo máximo posible como para poder analizar y evaluar su rendimiento y consumo energético.