

Arquitectura paralela y clusterización de muestras genéticas

Aingeru García
agarcia383@ikasle.ehu.eus

Ivan Calvo
icalvo037@ikasle.ehu.eus

Índice

1. Introducción	2
2. system.init	3
2.1. Máquinas utilizadas	3
2.2. Estructura del proyecto y máquina de desarrollo	4
2.3. Herramientas	5
2.3.1. GCC y OpenMP	5
2.3.2. CMake	5
2.3.3. Editores/IDEs	8
2.3.4. GitHub	9
2.4. Fundamentos teóricos y contextualización.	9
2.4.1. Paralelización con OpenMP	10
2.4.2. Thread safe y race conditions	12
2.4.3. Base de datos de muestras de elementos patógenos	13
2.4.4. Algoritmo K-means clustering	13
3. system.build	14
3.1. Versión en serie	14
3.1.1. Distancia genética	14
3.1.2. Nearest Cluster - grupo más cercano	15
3.1.3. Silhouette	16
3.1.4. Análisis de Enfermedades	18
3.1.5. Algoritmo Quicksort.	19
3.2. Versión en paralelo	20
3.2.1. gengrupos.c	21
3.2.2. fun.c	23
3.2.3. Distancia genética	23
3.2.4. Nearest Cluster - grupo más cercano	23
3.2.5. Silhouette	25
3.2.6. Análisis de Enfermedades	27
3.2.7. Inicializar centroides	29
3.2.8. Nuevos Centroides	30
4. system.metrics	32
4.1. Fundamentos teóricos	32
4.1.1. Incremento de velocidad	32
4.1.2. Time-slicing: más threads por núcleo	33
4.1.3. Factor de aceleración y eficiencia	34
4.2. Análisis de tiempos y eficiencia	35
4.2.1. Best record	35
4.2.2. Tablas de Tiempos, factor de aceleración y eficiencia	36
4.2.3. Graficas de los resultados	37
4.2.4. Observaciones y conclusiones	38
5. Reflexión	40
6. Bibliografía	41

1. Introducción

En este proyecto se aborda el desarrollo de un sistema multiprocesador para el procesamiento y análisis de grandes cantidades de datos genéticos utilizando el lenguaje de programación C y el API de OpenMP. El objetivo principal es lograr una mayor eficiencia y rapidez en el análisis de estos datos mediante la paralelización del algoritmo K-means clustering, utilizado para agrupar y clasificar los datos genéticos en diferentes categorías.

Para alcanzar este objetivo, nos nutrimos de unas bases de datos de muestras genéticas basadas en patógenos obtenidos en los laboratorios de la OMS y se ha desarrollado un programa que gestiona de manera eficiente el procesamiento y análisis de estos datos utilizando técnicas de paralelización. El resultado final es un sistema que permite realizar el análisis de grandes cantidades de datos genéticos, y así contrastar con su homólogo en serie/secuencial.

Esperamos que este proyecto sea el primer paso para poder crear aplicaciones que resulten de gran utilidad en el ámbito real y que requieran la paralelización de procesos para lograr una mayor optimización. La rapidez y eficiencia en el procesamiento y análisis de grandes cantidades de datos es crucial en muchos campos, como las finanzas, el análisis de datos de tráfico, el procesamiento de imágenes y video, entre otros. El desarrollo de este tipo de sistemas puede tener un impacto significativo en muchas áreas.



Figura 1: Genetics.

2. system.init

Antes de comenzar a trabajar en el proyecto, es importante establecer el contexto en el que se va a ejecutar.

Para determinar el entorno de ejecución del mismo, es necesario considerar aspectos como el sistema operativo, el hardware, las herramientas y librerías necesarias, entre otros. Además, es importante tener en cuenta cualquier limitación o restricción que pueda afectar la ejecución del programa, así como que las dependencias estén disponibles y se encuentren en la versión adecuada.

2.1. Máquinas utilizadas

Aunque el desarrollo del proyecto ha sido llevado a cabo en portátil y/o sobremesa, las pruebas han sido ejecutadas primordialmente en el **clúster** de la Facultad de Informática de Donostia, dadas las posibles inconsistencias entre los diversos sistemas (versiones, aleatoriedad...etc).

Además, el clúster de la Facultad nos proporciona un entorno de pruebas de alta capacidad y confiabilidad, lo que nos permite evaluar el rendimiento y la escalabilidad del proyecto.

Veamos las dos máquinas principales para las pruebas:

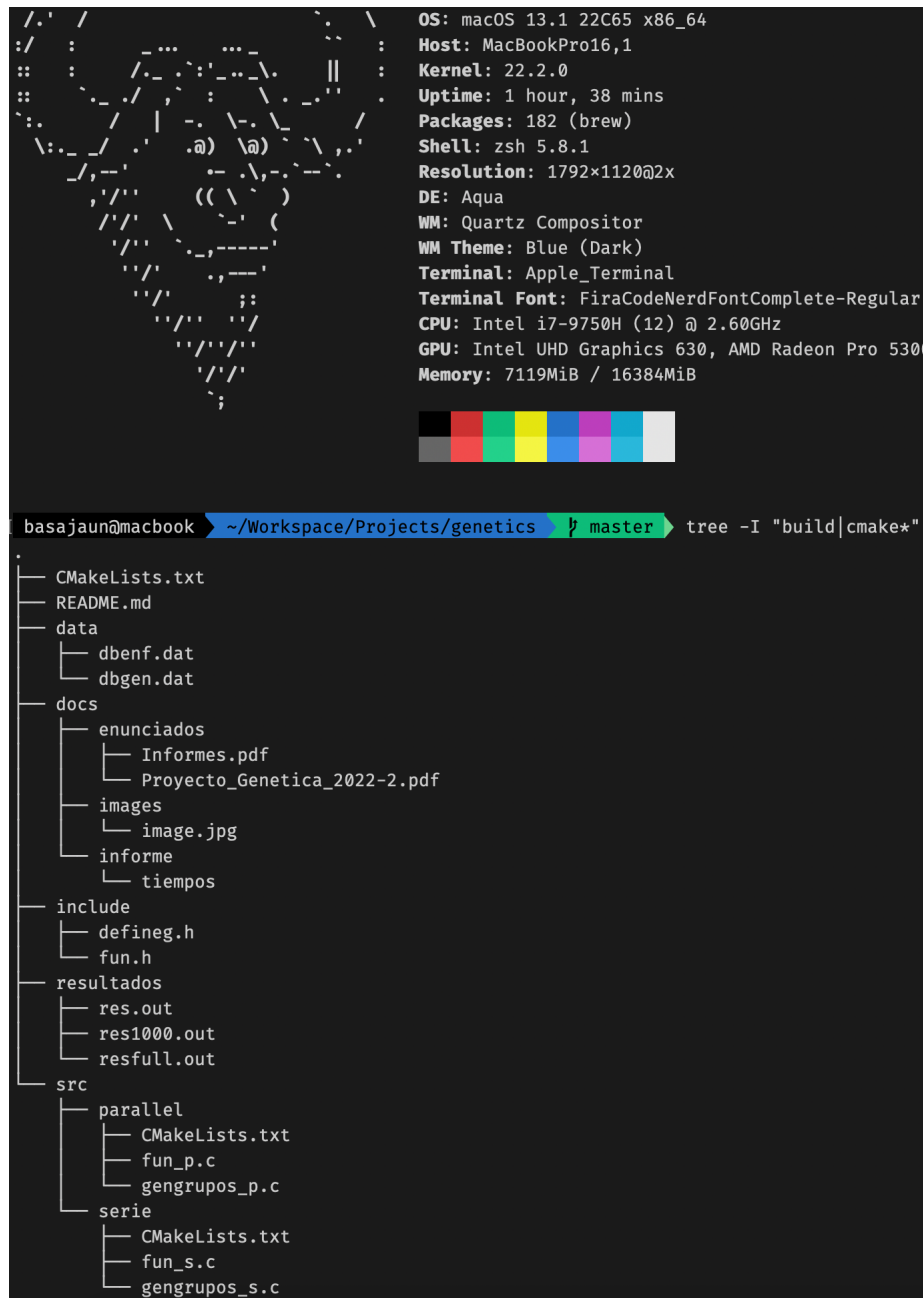
PC sobremesa

- S.O: GNU/Linux Debian 11 Bullseye
- CPU: CPU Intel Core i9 7920X 2.90 GHz - Cores 12 Threads 24
- Grafica: SLI (paralelo) 2 x GeForce GTX 1080 Ti
- RAM: 32.0GB @ 9013MHz
- SSD/HD: 476GB Samsung SSD 960 PRO 512GB

Clúster

- S.O: Rocks 7.0 Manzanita (CentOS 7.4)
- CPU: 2 procesadores Intel Xeon Gold 6130, 16 núcleos, 2,1 GHz 32 núcleos (64 hilos)
- RAM: 32 GB RAM (RDIMM - 2666 MT/s)
- INS: instrucciones vectores de 512 bits (AXV512)
- Gráfica: tarjeta gráfica NVIDIA QUADRO P4000

2.2. Estructura del proyecto y máquina de desarrollo



The screenshot shows a terminal window with a dark background. On the left, there is a large ASCII art logo of a cat. On the right, system information is displayed in a light blue font. Below this, a command is executed to show the directory tree of a project.

```
OS: macOS 13.1 22C65 x86_64
Host: MacBookPro16,1
Kernel: 22.2.0
Uptime: 1 hour, 38 mins
Packages: 182 (brew)
Shell: zsh 5.8.1
Resolution: 1792x1120@2x
DE: Aqua
WM: Quartz Compositor
WM Theme: Blue (Dark)
Terminal: Apple_Terminal
Terminal Font: FiraCodeNerdFontComplete-Regular
CPU: Intel i7-9750H (12) @ 2.60GHz
GPU: Intel UHD Graphics 630, AMD Radeon Pro 530
Memory: 7119MiB / 16384MiB
```

```
basajaun@macbook ~/Workspace/Projects/genetics master tree -I "build|cmake*"
.
├── CMakeLists.txt
├── README.md
├── data
│   ├── dbenf.dat
│   └── dbgen.dat
├── docs
│   ├── enunciados
│   │   ├── Informes.pdf
│   │   └── Proyecto_Genetica_2022-2.pdf
│   ├── images
│   │   └── image.jpg
│   └── informe
│       └── tiempos
├── include
│   ├── defineg.h
│   └── fun.h
├── resultados
│   ├── res.out
│   ├── res1000.out
│   └── resfull.out
└── src
    ├── parallel
    │   ├── CMakeLists.txt
    │   ├── fun_p.c
    │   └── gengrupos_p.c
    └── serie
        ├── CMakeLists.txt
        ├── fun_s.c
        └── gengrupos_s.c
```

Figura 2: Máquina de desarrollo, estructura de proyecto y saludos de GNU.

2.3. Herramientas

Aquí presentamos algunas de las herramientas más significativas de las que hemos utilizado.

2.3.1. GCC y OpenMP

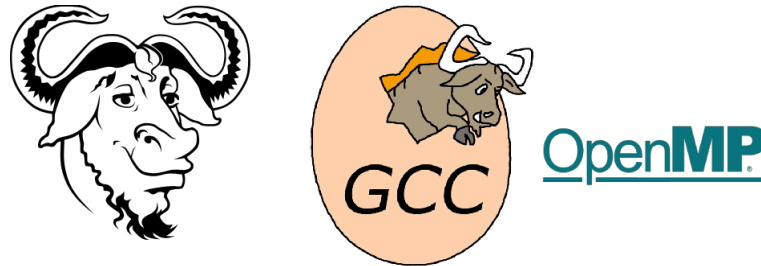


Figura 3: Compilador gcc y OpenMP

La versión que actualmente ejecuta el clúster mediante su link simbólico es la versión 7.2, que apunta a `/opt/gnu/gcc/bin`. Sin embargo, el software de automatización de construcción de software que hemos utilizado - **CMake**, tiene vinculado una versión mucho más antigua. Dicha versión incluye una revisión de **OpenMP** inferior a la 4.5, que es la que necesitamos para poder aplicar ciertas cláusulas como es la de `reduction` a los arrays. Así que en el siguiente apartado veremos cómo ajustarlo.

2.3.2. CMake

Aunque podríamos haber utilizado otras herramientas con las que estamos familiarizados para automatizar tanto el proceso de build/compilación como la ejecución de la aplicación, como por ejemplo `make` (en *crudo*) o `shell scripting`, hemos terminado decidiéndonos por utilizar **CMake**.

Lo consideramos una muy buena opción ya que de manera subyacente hace uso de `make` y, como se verá un poco más adelante, si se configura debidamente, hoy en día dispone de una maravillosa integración con editores de texto y/o IDEs modernos, lo cual agiliza mucho el proceso de desarrollo.

En cuanto a su funcionamiento y uso, está muy extendido en todo tipo de sistemas y entornos. No obstante, en los sistemas operativos Unix el sistema de compilación nativo suele ser un conjunto de archivos **Makefile**. La herramienta de compilación `make` utiliza los mismos para transformar los ficheros fuente en ejecutables y bibliotecas. **CMake** ofrece una forma de abstraer el proceso de generación de estos los ficheros Makefile en un lenguaje de programación independiente del sistema. De esta manera, se puede utilizar CMake para generar los ficheros necesarios para la compilación en sistemas Unix (GNU/Linux y MacOS) de manera más sencilla y genérica.

Imagen clarificativa del funcionamiento de CMake.

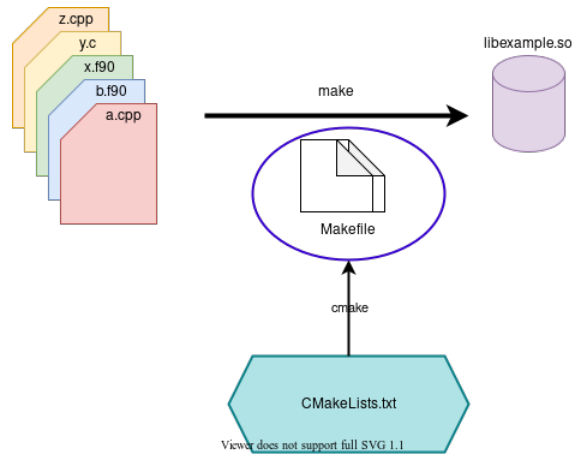


Figura 4: Funcionamiento de CMake [3]

Veamos el código que hemos preparado para nuestro proyecto, se puede hallar en el fichero `CMakeLists.txt` en el directorio `root` del mismo. Nótese la línea de código `set(CMAKE_C_COMPILER /opt/gnu/gcc/bin/gcc)`, arregla lo mencionado antes - decimos de manera explícita que compilador utilizar, ya que CMake, en el clúster está vinculado a una versión mucho más antigua de `gcc`. Utilizamos el nivel de optimización 2.

```
cmake_minimum_required(VERSION 3.12.1)
set(CMAKE_C_COMPILER /opt/gnu/gcc/bin/gcc)
project(genetics C)

set(CMAKE_C_STANDARD 17)

include_directories(include)

add_subdirectory(src/serie/
                 src/parallel/)

add_executable(genetics_s
               src/serie/fun_s.c
               src/serie/gengrupos_s.c)

add_executable(genetics_p
               src/parallel/fun_p.c
               src/parallel/gengrupos_p.c)

find_package(OpenMP)
if (OPENMP_FOUND)
    set (CMAKE_C_FLAGS "-O2${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set (CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${OpenMP_EXE_LINKER_FLAGS}")
endif()

target_link_libraries(genetics_s m)
target_link_libraries(genetics_p m)
```

Para sistemas MacOS X, se puede agregar la siguiente linea de código:

```
if (OPENMP_FOUND)
    # La flag OpenMP_C_FLAGS contiene -Xclang en MacOS, fuerzo su eliminación
    # a la hora de pasarle las flags de OpenMP al compilador.
    string(REPLACE "-Xclang" "" OpenMP_C_FLAGS ${OpenMP_C_FLAGS})
    set (CMAKE_C_FLAGS "-O2${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
    set (CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${OpenMP_EXE_LINKER_FLAGS}")
endif()
```

Nótese que si se gestiona a través de IDE o de manera incorrecta la inicialización de CMake, es posible que se establezca el 'target type' a **DEBUG**, con lo que conviene o bien configurarlo desde el IDE correctamente o estableciendo el modo que se desea a través de la terminal en cualquier sistema Unix:

```
mkdir Release
cd Release
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

Si se desea utilizar el debugger, el target type debe de estar en **DEBUG**, para que el compilador pueda agregar los símbolos y breakpoints oportunos. Sin embargo, se desea realizar pruebas de velocidad, es fundamental que la aplicación esté en **Release** o al menos en **RelWithDebInfo** (Release with Debug Info), que no sacrifica rendimiento y agrega parcialmente información local de stack en los crash dumps, por ejemplo.

Durante el desarrollo - por agilidad - nosotros disponíamos de una versión **DEBUG** y otra **RELEASE** de lo siguiente:

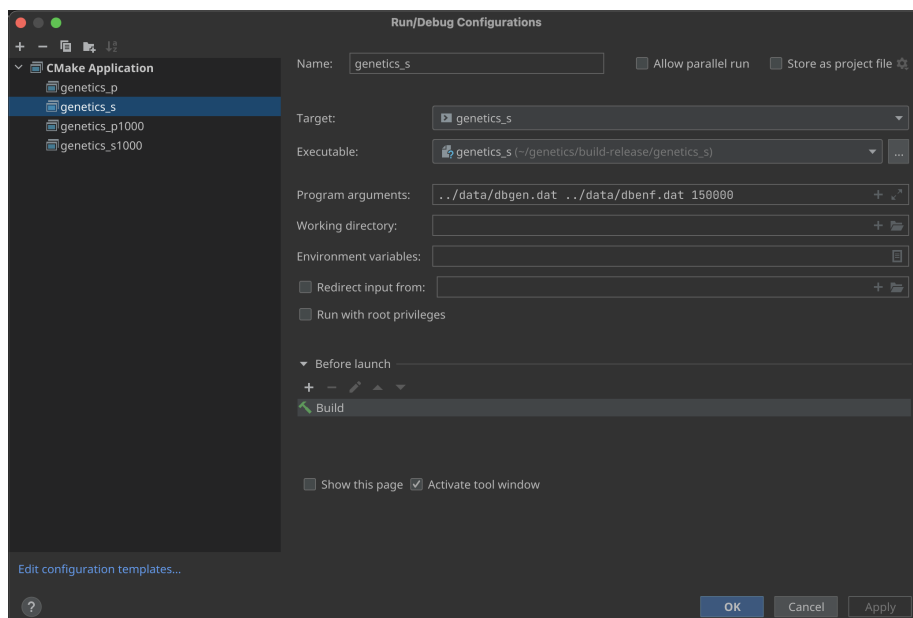


Figura 5: Build/Run configurations

De manera que, en conjunción con CMake, con tan sólo una determinada combinación de teclas o *keybind* podemos lanzar la versión que deseemos. Favoreciendo la fase de testing. Ésto se podría también haber conseguido simplemente con la creación de un script `.sh` y alias.

2.3.3. Editores/IDEs

Para trabajar en el clúster se recomendaba el uso directo del editor Nano, el cual está muy bien y conocíamos. En general, estamos acostumbrados a utilizar una versión moderna de Vim - NeoVim [1], el cual permite su configuración mediante plugins escritos en LUA. Tuvimos problemas para poner nuestra configuración personal en el clúster al no poder instalar software, así que decidimos explorar algún IDE moderno.

Para el desarrollo en profundidad, nos hemos aventurado con un nuevo IDE, el cual hemos conseguido en su versión *ultimate* gracias a ser alumnos de la EHU/UPV: **Clion** [2]. Como al ejecutarlo en nuestros sistemas obteníamos distintos resultados dadas las diferentes versiones de compiladores e incluso funciones como `rand()` de C, hemos utilizado un plugin de Clion para poder trabajar de manera remota mediante SSH en el mismo IDE directamente, ya que establece un backend en el clúster:

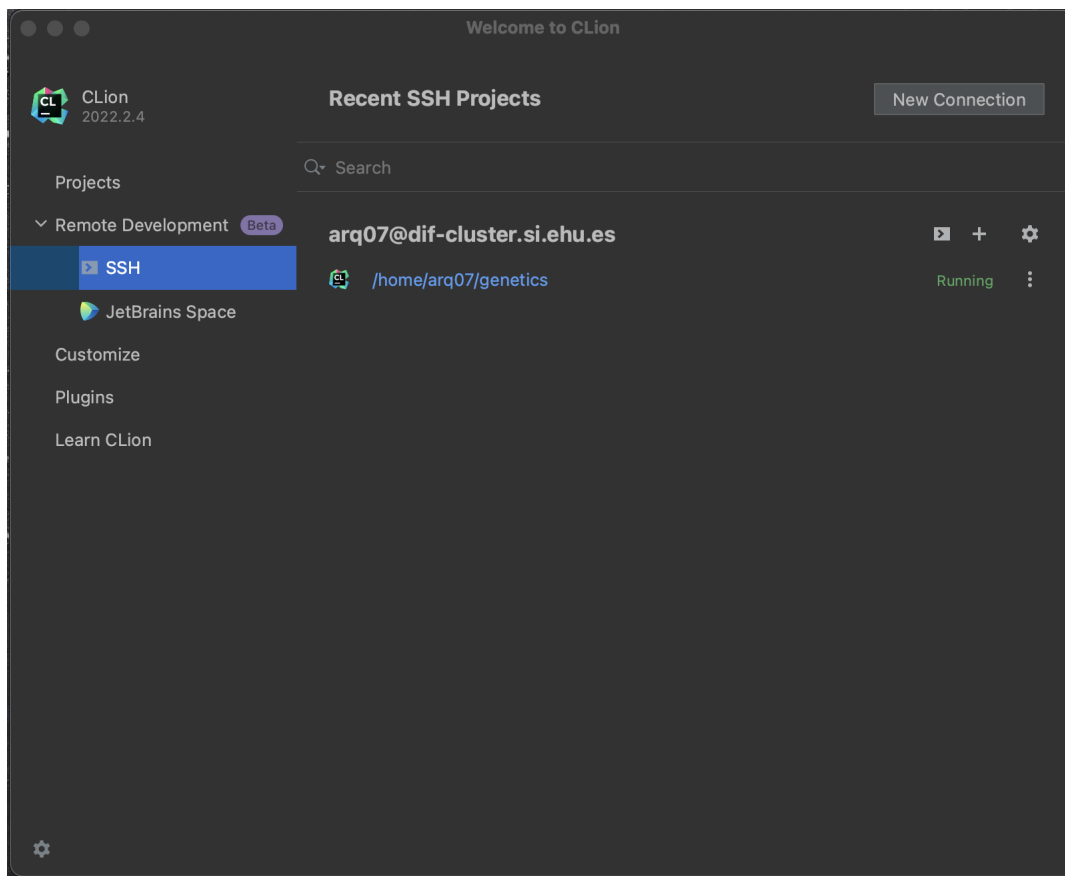


Figura 6: IDE Backend con SSH

De manera adicional, para saber si era buen momento para lanzar nuestras ejecuciones, nos permitía ver la carga en el clúster:

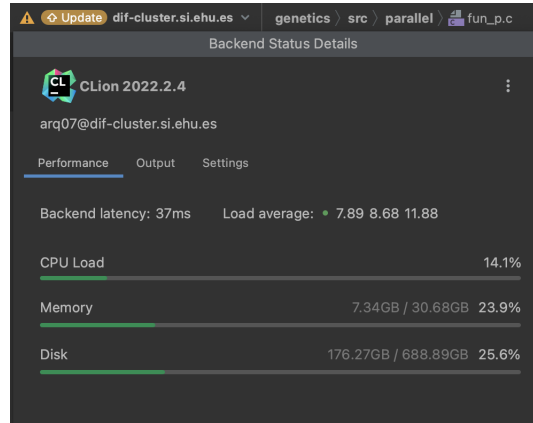


Figura 7: Status del clúster, con alertas dependiendo del nivel de carga.

Aunque esto se pueda consultar con comandos como `top`, favorece la comodidad desde el entorno de desarrollo y, siempre conviene aventurarse y aprender cosas nuevas.

2.3.4. GitHub

Sabemos que no es estrictamente necesario, pero consideramos fundamental para cualquier proyecto ya sea en solitario o colaborativo el uso de sistemas de control de versiones como Git y la plataforma con más fama como es GitHub. Una vez te acostumbras, es difícil no aplicarlo a cualquier proyecto que se desarrolle.

Como nota en el clúster necesitamos arrancar primero el ssh-agent para poder así crear las claves SSH y así posteriormente vincularlas a la cuenta/perfil de trabajo.

[Repositorio del proyecto.](#)

2.4. Fundamentos teóricos y contextualización.

Como se ha comentado previamente, el objetivo de este proyecto es el de trabajar la paralelización de una aplicación en un sistema multiprocesador para poder mejorar la eficiencia del mismo. Para lograr esto, es importante tener una comprensión sólida de cómo funcionan los datos en su contexto (es decir, en el ámbito de la genética y del algoritmo de K-means clustering) y de los conceptos básicos de paralelización en general, así como de la utilización de la API de OpenMP en particular.

2.4.1. Paralelización con OpenMP

OpenMP es una biblioteca de programación paralela que ofrece una amplia gama de funciones para facilitar la implementación de código concurrente en aplicaciones. Un ejemplo de esta es la función `set_omp_num_threads()`, que permite establecer el número de hilos que se deben utilizar para ejecutar una tarea en paralelo. Además de esta función, OpenMP ofrece otras como `omp_get_num_threads()`, `omp_get_thread_num()` y `omp_get_max_threads()`, entre muchas otras.

Sin embargo, en este caso nos hemos centrado en el uso de las directivas de OpenMP, que se indican mediante el uso del *pragma*. Estas directivas nos permiten establecer regiones del código que deben ejecutarse en paralelo siguiendo el **modelo FORK-JOIN**. Esto significa que se crean copias de un hilo, conocidos como hilos esclavos, que se encargan de realizar una tarea en paralelo con el hilo principal o máster. Una vez que los hilos esclavos o slaves han finalizado su tarea, se reúnen con el hilo máster para continuar la ejecución del código en serie. Como pequeña nota sobre la nomenclatura utilizada, hoy en día se está intentando evitar estos nombres debido a su carga simbólica negativa y el potencial de alienar u ofender a ciertos grupos de personas.

La paralelización se refiere a la distribución de la carga de un proceso en varios hilos de ejecución, de manera que se pueda reducir el tiempo necesario para procesar la misma cantidad de datos o realizar una operación en particular. Esto se puede lograr distribuyendo la carga de trabajo (conocida como *load balancing*) de manera apropiada entre los hilos de ejecución (en algunos casos de manera equitativa, pero existen diferentes técnicas para abordar esto, que más adelante veremos).

Para paralelizar una aplicación utilizando OpenMP, debemos definir las regiones del código que queremos ejecutar de manera paralela y luego indicar a cada hilo qué parte del código debe ejecutar. Esto se puede hacer de manera manual o automática (con la ayuda de OpenMP).

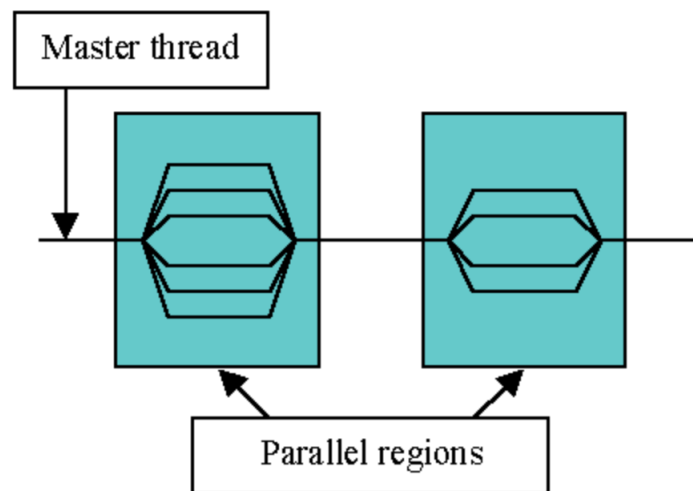


Figura 8: OpenMP threading.

Veamos con un ejemplo básico para comprender ésta idea general sobre cómo se puede repartir el trabajo entre varios procesos:

```
int main() {
    int nthreads, tid, i;
    double suma = 0.0;
    double a[100];
    #pragma omp parallel num_threads(4) private(nthreads, tid)
    {
        tid = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        #pragma omp for
        for (i = tid; i < 100; i += nthreads) {
            suma += a[i];
        }
    }
    printf("La suma total es: %f\n", suma);
    return 0;
}
```

En este ejemplo, cada hilo realizará la suma de un conjunto de elementos del array `a` y el resultado final será la suma total de todos los elementos del array. El uso de `reduction` haría ésta operación más eficiente, pero de ésta manera se puede ver el reparto de trabajo intercalado o por bloques entre hilos.

En general, los conceptos clave que debemos tener en cuenta al desarrollar una aplicación paralela, se encuentran el uso de hilos y regiones paralelas, el manejo de la carga de trabajo y el uso (en nuestro caso) de la API de OpenMP para controlar la paralelización.

Veamos brevemente una directiva que encapsula varios de los conceptos que de manera recurrente podremos encontrarnos a lo largo del proyecto:

```
#pragma omp for
```

OpenMP nos proporciona ésta directiva la cual ha de estar dentro de una región paralela (`#pragma omp parallel`) y se encarga de gestionar el reparto de trabajo de cada uno de los hilos de manera automática. El número de hilos se puede determinar de varias formas, por ejemplo: mediante `set_num_threads()`, `EXPORT` o incluso como cláusula adicional para el `for` o región paralela.

También nos permite indicarle qué tipo de distribución o reparto de trabajo queremos que haga, mediante las diferentes opciones que acepta la cláusula `schedule`, como *static*, *guided*, y *dynamic*. Nótese que si no se indica nada, su planificación por defecto será *static*.

La siguiente imagen sirve de ayuda para visualizar el funcionamiento y reparto de trabajo de esta directiva:

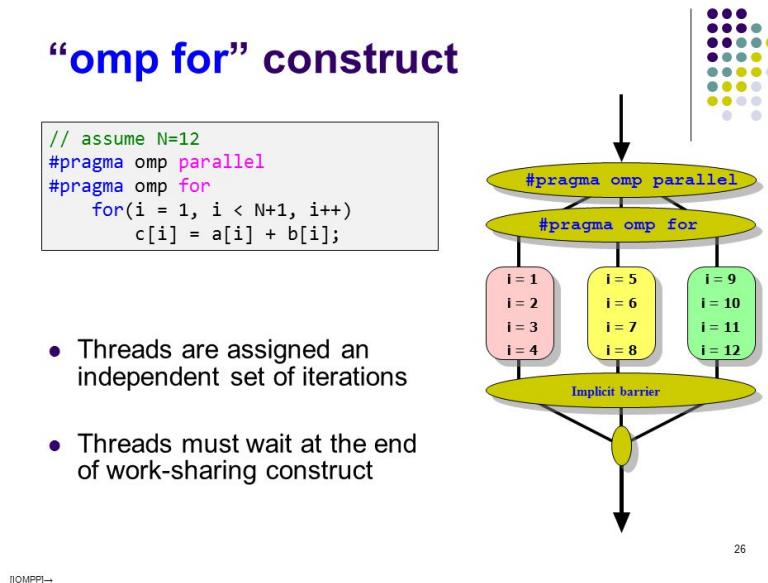


Figura 9: `#pragma omp for`

Las variables definidas dentro del bucle `for` son locales al hilo que las ejecuta, lo que significa que cada hilo tiene su propia copia de las variables. Por lo tanto, es importante tener en cuenta que cada hilo puede tener un valor diferente para la misma variable.

2.4.2. Thread safe y race conditions

El concepto de **thread safe** se refiere a la capacidad de un dato para ser modificado por múltiples hilos al mismo tiempo sin causar errores o **race conditions**. Esto se logra mediante el uso de mecanismos de sincronización, como `locks` u operaciones atómicas, que bloquean el acceso a los datos para evitar la interacción conflictiva entre hilos. Forzando a que el resto esperen a que se haga `unlock`.

Las condiciones de carrera o **race conditions**, son un problema importante y difícil de depurar o debuggear que surge cuando distintos hilos tienen acceso simultáneo a la misma posición de memoria. Esto puede llevar a resultados impredecibles o inconsistentes, ya que cada hilo tomará una copia del valor de la posición en un momento determinado y (hará un *snapshot*), al intentar modificarlo y dejarlo de nuevo en la posición original, pueden sobrescribirse mutuamente. Para ello existen también mecanismos conocidos como `mutex` (mutual exclusion).

Al tratar de paralelizar un sistema, es crucial tener cuidado con este tipo de problemas y estar atentos a cualquier posible ocurrencia de race conditions en la memoria compartida.

2.4.3. Base de datos de muestras de elementos patógenos

La aplicación de éste proyecto es la de realizar un Análisis genético de muestras de elementos patógenos obtenidos en los laboratorios de la OMS. Para ello se nos han proporcionado dos ficheros que tienen la función de base de datos genéticas distribuidos de la siguiente manera:

- **dbgen.dat**: 211k muestras de diferentes pacientes con 40 características genéticas cada una de ellas.
- **dbenf.dat** Fichero que, para cada una de las muestras del archivo anterior, expresa las probabilidades de que dicho paciente adquiera las 18 posibles enfermedades especificadas, en base a las características genéticas del mismo.

2.4.4. Algoritmo K-means clustering

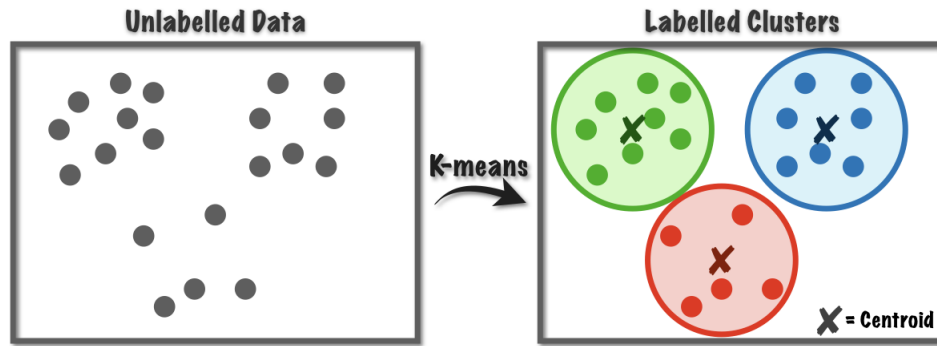


Figura 10: K-means clustering

El algoritmo k-means es un método de aprendizaje no supervisado utilizado para clasificar un conjunto de datos en k grupos, basándose en la similitud entre ellos. La forma en la que funciona es la siguiente:

- 1. Inicialmente, se eligen k puntos al azar del conjunto de datos como centroides de cada uno de los k grupos. Siendo k el hiperparámetro del algoritmo.
- 2. A continuación, se asigna cada punto del conjunto de datos al grupo cuyo centroide esté más cerca.
- 3. Una vez asignados todos los puntos a un grupo, se recalculan los centroides de cada grupo como la media de los puntos que pertenecen a ese grupo.
- 4. Se repiten los pasos 2 y 3 hasta que los centroides de los grupos ya no cambien o hasta que se alcance un número máximo de iteraciones. En cada ciclo, se comprobará la calidad o bondad de la partición o distribución de clústers mediante su **intra-clúster** (distancia media de entre todos los elementos del clúster) e **inter-clústers** (distancia media de entre todos centroides de los clústers).

Una vez finalizado el algoritmo, cada punto del conjunto de datos estará asignado a uno de los k grupos y cada uno de estos grupos estará definido por su centroide y los puntos que lo forman.

Nótese que el algoritmo K-means clustering **no garantiza encontrar la solución óptima**, ya que depende de la elección inicial de los centroides. Por lo tanto, en el código base que se nos ha proporcionado, ya existen las funciones y sistemas que se encargan de estas iteraciones. Además, el algoritmo k-means es sensible a los valores atípicos, por lo que es importante tratar de identificarlos y eliminarlos antes de aplicar el algoritmo.

3. system.build

A lo largo del proyecto, hemos realizado una serie de tareas y hemos ido avanzando en su desarrollo. En lugar de presentar solo la versión final, consideramos importante el también proporcionar información adicional que describa el proceso de trabajo y nos permita revisitar y documentar parte de los errores.

En este sentido, vamos a exponer cómo se han ido llevando a cabo las distintas tareas y cómo hemos llegado a las conclusiones finales. Además, también incluiremos información sobre algunos errores que hemos ido cometiendo a lo largo del proceso, ya que consideramos que este tipo de reflexión es esencial para el aprendizaje y el refuerzo continuo.

3.1. Versión en serie

En esta sección, vamos a explicar cómo implementar algunas funciones para cumplir con el propósito de la aplicación. Hay que tener en cuenta que, al trabajar en serie, el modelo utilizado para estas implementaciones se basará en una ejecución **mono-hilo**, con lo que se procesará de manera secuencial y no se beneficiará de la paralelización.

3.1.1. Distancia genética

Para la distancia genética tenemos el siguiente código para simplemente recorrer todos los elementos de ambos vectores mientras calculamos la distancia Euclídea de entre todos los elementos de la misma posición.

Código:

```
double geneticdist(float *elem1, float *elem2)
{
    double total = 0.0f;

    for(int i = 0; i < NCAR; i++)
        total += pow((double)(elem2[i] - elem1[i]), 2);

    return sqrt(total);
}
```

Explicación:

Primera y más simple función implementada, cuya utilidad es la de devolver un dato de tipo `double` que represente la distancia Euclídea entre dos vectores al comparar sus 40 características genéticas. Esto se consigue con el cuadrado de la resta de cada una de las características genéticas de ambos vectores (los cuales representan muestras). Esto es, la siguiente fórmula:

$$d(\mathbf{p}, \mathbf{q}) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2}$$

3.1.2. Nearest Cluster - grupo más cercano**Código:**

```
void nearest_cluster(int nelem, float elem[][NCAR], float cent[][NCAR], int *samples)
{
    double dist, mindist;

    for(int i = 0; i < nelem; i++){
        mindist = DBL_MAX;
        for(int k = 0; k < nclusters; k++) {
            dist = geneticdist(elem[i], cent[k]);
            if(dist < mindist){
                mindist = dist;
                samples[i] = k;
            }
        }
    }
}
```

Explicación:

Función encargada de asociar mediante el array denominado como `samples` el grupo más cercano a cada una de las muestras. Para ello, se recorren todas las muestras contrastando cual de los centroides que representan a cada uno de los clústeres cercanos posee la menor de las distancias genéticas a la misma. Una vez computado, se procede a la asociación mediante el array `samples`.

3.1.3. Silhouette

Código:

```
double silhouette_simple(float samples[][NCAR], struct lista_grupos *cluster_data,
                        float centroids[][NCAR], float a[]){

    float b[nclusters];
    double tmp;

    for(int k = 0; k < nclusters; k++) b[k] = 0.0f;
    for(int k = 0; k < MAX_GRUPOS; k++) a[k] = 0.0f;
    // Me baso en teoría de grafos para obtener el peso total de las distancias
    // Según se va avanzando, únicamente tengo en cuenta las distancias
    // con elementos posicionados en posiciones mayores que la actual.
    for(int k = 0; k < nclusters; k++){
        tmp = 0;
        for(int i = 0; i < cluster_data[k].nelems; i++){
            for(int j = i + 1; j < cluster_data[k].nelems; j++){
                tmp += geneticdist(samples[cluster_data[k].elem_index[i]],
                                   samples[cluster_data[k].elem_index[j]]);
            }
        }

        // medidas para los n elementos de cada clúster. n(n-1)/2.
        // Equivale al Cálculo de aristas totales para un grafo completo.
        float narista = ((float)(cluster_data[k].nelems *
                                   (cluster_data[k].nelems - 1)) / 2);
        a[k] = cluster_data[k].nelems <= 1 ? 0 : (float)(tmp / narista);
    }

    // aproximar b[i] de cada cluster
    for(int k = 0; k < nclusters; k++){
        for(int j = 0; j < nclusters; j++){
            b[k] += (float) geneticdist(centroids[k], centroids[j]);
        }
        b[k] /= (float)(nclusters - 1);
    }

    float max, sil = 0.0f;
    for(int k = 0; k < nclusters; k++){
        max = a[k] >= b[k] ? a[k] : b[k];
        if(max != 0.0f)
            sil += (b[k] - a[k]) / max;
    }

    return (double)(sil / (float)nclusters);
}
```

Explicación:

Esta función tendrá **gran impacto** en los tiempos de la clusterización, con lo que se le deberá de prestar especial atención durante el proceso de paralelización.

El propósito de ésta función es el de calificar la calidad de una partición o distribución de clústeres, lo cual servirá para determinar si el número k de clústers elegido es óptimo o, por el contrario, se ha de continuar iterando.

Para realizar ésta calificación, el algoritmo se basa en los dos *CVI* (Clúster validation index). Para medir cuan compacto es el clúster, se miden las distancias genéticas de entre todas las muestras que constituyan el mismo y se calcula su media. Ésto lo almacenamos en el array a , cuyo índice k indicará el número de clúster y el contenido - la densidad.

Ésto lo denominamos como **densidad intra-clúster** y, aunque existen otras maneras de hacerlo, nos hemos basado en teoría de grafos para su cálculo - tratando cada muestra como un vértice y las distancias entre una muestra y otra como aristas. Con lo que, para computar la media tratamos el clúster como un **grafo completo** - para todos sus vértices, analizamos y acumulamos la distancia genética con los vértices **restantes** (no los ya analizados, de ahí el $j + 1$ del **for** interno). A la suma de todas las distancias, la dividimos entre el número de distancias (o aristas) totales, que para un grafo completo vienen determinadas por la fórmula:

$$\frac{n(n-1)}{2}$$

Una vez almacenado la **densidad intra-clúster** en el array a , procedemos al cálculo del segundo *CVI*, la **distancia intra-clúster**.

Para ello, simplemente almacenamos en el array b , la media de las distancias genéticas del centroide que representa cada clúster, al resto de centroides.

Una vez computados los dos *CVI*, procedemos al cálculo del factor s ó *calidad de partición de clústers*, que viene determinado por la siguiente fórmula:

$$s[k] = \frac{b_k - a_k}{\max[a_k, b_k]}$$

Para después devolver el valor de:

$$\frac{\sum_{n=1}^k s[n]}{|k|}$$

3.1.4. Análisis de Enfermedades

Código:

```
void analisis_enfermedades(struct lista_grupos *cluster_data, float enf[][TENF],
                          struct analisis *analysis)
{
    int cluster_size;
    float median;
    float *disease_data;

    for(int i = 0; i < TENF; i++){
        analysis[i].mmin = FLT_MAX;
        analysis[i].mmax = 0.0f;
    }

    for(int k = 0; k < nclusters; k++){
        cluster_size = cluster_data[k].nelems;

        for(int j = 0; j < TENF; j++){
            disease_data = malloc(sizeof(float) * cluster_size);
            for(int i = 0; i < cluster_size; i++) disease_data[i] = 0.0f;

            // Para cada enfermedad, recorro todos las muestras del clúster
            // recogiendo sus datos.
            for(int i = 0; i < cluster_size; i++)
                disease_data[i] = enf[cluster_data[k].elem_index[i]][j];

            // Tengo el array para ésta enfermedad en éste clúster.
            // Solo me queda ordenar y calcular la mediana.
            median = sort_and_median(cluster_size, disease_data);

            //Ya tengo la median para ésta enfermedad y éste clúster.
            if(median > analysis[j].mmax){
                analysis[j].mmax = median;
                analysis[j].gmax = k;
            }

            if(median > 0 && median < analysis[j].mmin){
                analysis[j].mmin = median;
                analysis[j].gmin = k;
            }

            free(disease_data); // Para cada enfermedad
        }
    }
}
```

Explicación:

En éste caso, ésta es la función encargada de asociar a un **struct** de enfermedades los datos más representativos de las mismas de entre todas las muestras encontradas en nuestra distribución de clústeres.

Para su análisis, simplemente recorremos todos los clústeres y calculamos la mediana que representa la enfermedad en cada uno de ellos. Para ello, cada vez que se analiza un clúster, se genera una lista para cada enfermedad con las probabilidades cada una de las muestras que constituyen dicho clúster. Ésta información la almacenamos en la memoria dinámica o heap, ya que su tamaño variará en cada iteración. Esa lista se ordena mediante el algoritmo Quicksort en su versión iterativa, y es entonces cuando se selecciona la mediana de dicha lista - es esa mediana la encargada de representar a ese grupo o clúster para esa enfermedad.

De ésta manera, podemos contrastar si es una 'máxima mediana' - o bien la mayor, o bien la menor. En caso afirmativo, actualizamos el struct `analysis` con dicho dato y clúster en el que se encuentra.

3.1.5. Algoritmo Quicksort.

En una primera instancia implementamos el algoritmo que se sugería en el enunciado, el de la burbuja. Sin embargo con afán de optimizar un poco más, lo sustituimos por el algoritmo **Quicksort** pero en su forma recursiva [4], para luego optimizarlo más aún con su versión iterativa y un par de cambios. Veamos el algoritmo utilizado:

Código:

```
void swap(float *a, float *b) {
    float t = *a;
    *a = *b;
    *b = t;
}

int partition(float array[], int low, int high) {
    float pivot = array[high];

    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            i++;
            swap(&array[i], &array[j]);
        }
    }

    swap(&array[i + 1], &array[high]);
    return (i + 1);
}

void quickSort(float array[], int low, int high) {
    while (low < high) {
        int pivotIndex = partition(array, low, high);
        quickSort(array, low, pivotIndex - 1);
        low = pivotIndex + 1;
    }
}
```

Simplificando, el algoritmo QuickSort funciona de la siguiente manera:

Algoritmo Quicksort

- 1) Inicializar dos punteros izquierda y derecha al primer y último elemento
- 2) Mientras izquierda sea menor que derecha, dividir el array en dos mitades usando un pivot y llamar recursivamente a la función quickSort con la primera mitad. Actualizar izquierda para apuntar al primer elemento de la segunda mitad.
- 3) Llamar recursivamente a la función quickSort con el array entre izquierda y derecha para ordenar cada una de las dos mitades del array.

Complejidad/Orden:

- **Worst:** $\mathcal{O}(n^2)$ | **Average:** $\mathcal{O}(n \log n)$ | **Best:** $\mathcal{O}(n \log n)$

3.2. Versión en paralelo

Para la versión en paralelo, hemos intentado aplicar diversas directivas y utilizar de una manera extendida lo aprendido de **OpenMP**, sin embargo, según hemos ido trabajando en ello han aparecido maneras más sencillas en las que los resultados mejoraban - con lo que, hemos optado por abarcar menos.

En éste fase, nos centramos en paralelizar las funciones implementadas por nosotros mismos y, de manera adicional, parte del código que se nos proporcionó de base.

En algunos casos nos habría gustado crear regiones paralelas más amplias y hacer uso de más directivas/funcionalidades de OpenMP, pero en función de cómo está construido el código y los problemas con los que nos hemos ido encontrando sobre la marcha, en algunos casos ha sido mejor opción el paralelizar de manera interna algunas funciones en lugar de crear regiones paralelas que las invoquen y optar por la sencillez. No obstante, nuestra postura ha sido la de intentar reducir el número de veces que se crean y destruyen hilos.

Tras mucho *jugar* con el código y realizar muchísimos tests comparando el rendimiento con y sin paralelizar, el código resultante es la versión con la que mejores resultados hemos obtenido. Cabe destacar, que en según que momentos ha sido un poco complicado el tomar datos para las métricas ya que, la máquina clúster, al ser de uso compartida, variaba mucho en tiempos en base a los recursos que se le estuviesen solicitando por otras personas.

Nosotros nos hemos decidido por establecer la cantidad de hilos en `gengrupos.c`.

Detalles, notas y problemas:

- **Race conditions:** nos forzaban a gestionar de una manera menos eficiente el código, con `pragma omp critical, single` etc. Pero a su vez, segura. Secciones candidatas a race conditions, han de ser desarrolladas con cuidado y poco a poco. Depurar esto es muy tedioso y complicado.
- **Ámbitos:** de algunas variables, que se necesitaban especificar de manera explícita en la región paralela de dentro de la función. Es algo que parece sencillo, pero en algún caso puede traer quebraderos de cabeza. No obstante, nos ha ayudado a visualizar el paralelismo entre los distintos procesos.
- **Reloj:** Otro problema que teníamos era el control del reloj y sus respectivas variables al crear regiones paralelas que abarcasen varias de las actuales. Esto lo solucionábamos indicando que únicamente el hilo máster era el encargado de gestionarlas. Aún así es un detalle que consideramos importante mencionar. No obstante, como se verá en su apartado correspondiente hemos considerado mejor opción su no paralelización.
- **Nowait:** Utilización recurrente de la cláusula `nowait` con objeto de mejorar el rendimiento del programa. Como en muchos casos no es necesario sincronizar datos al terminar un bucle, hemos optado por decirle a los hilos que al terminar su proceso no esperen al resto, y que sigan con las iteraciones asignadas por OpenMP. Ésto ha sido de gran utilidad en aquellos casos donde existían varios bloques de código sin dependencias entre sí, ergo no existía problemas de concurrencia entre variables compartidas. Ahora bien, como se verá más adelante - esto ha de usarse *con cabeza*, si no puede llevar a situaciones donde identificar el problema es muy complicado.
- **Planificación/Scheduling:** En OpenMP, al utilizar la directiva `#pragma omp for`, si se omite la cláusula `schedule` se asume se es consciente que se está utilizando un tipo de planificación **estática**. En la gran parte de la paralelización, es el tipo que hemos decidido era la mejor opción, ya que casi siempre se conoce en **en tiempo de compilación** el número de iteraciones y un reparto de trabajo **equitativo** de las mismas es conveniente. No obstante, hemos introducido tan sólo un bucle como **dinámico**, como se verá más adelante. Ya que tanto el trabajo como el número de iteraciones que se van a realizar puede ser desconocido.

3.2.1. gengrupos.c

Durante el desarrollo de la versión paralela, como se puede apreciar en los cambios que se han ido realizando en GitHub, han existido diversas versiones paralelizadas de éste fichero. Pero, consideramos que la versión final es la que mejores resultados proporciona.

Se podría decir que, de manera independiente, tan sólo existe una fracción de código que requiere paralelización para mejorar drásticamente las métricas ya que al ser una función de **complejidad lineal**, se iterará tantas veces como muestras existan, lo cual es posible sean números muy altos.

Código:

```
#pragma omp parallel default(none) private(i, grupo),\
    shared(cluster_data, popul, nclusters, nelem)
{
#pragma omp for
    for (i = 0; i < nclusters; i++) cluster_data[i].nelems = 0;

#pragma omp for private(grupo, i)
    for (i = 0; i < nelem; i++) {
        grupo = popul[i];
#pragma omp critical
        {
            cluster_data[grupo].elem_index[cluster_data[grupo].nelems] = i;
            cluster_data[grupo].nelems++;
        }
    }
}
```

Especificación:

Región paralela: `#pragma omp parallel for default(none):`

- `shared(cluster_data, popul, nclusters, nelem)`

Peligro en concurrencia: `cluster_data`, `popul` son accedidas por distintos hilos en la región paralela.

- `private(i, grupo)`

Motivo: Nótese que `i`, al estar declarada fuera de la región paralela y no en el `for` como es habitual tiene que ser definida explícitamente como `private`. Por otro lado, `grupo` depende del índice `i` y cada hilo necesita acceder en base a éste índice ya que lo que necesitamos es el grupo de una muestra en cada iteración.

- **Thread-safe check:** En este caso es muy sencillo que existan **race conditions**: aunque `i` no pueda tener el mismo valor para múltiples hilos, e incluso `grupo` sea privada, ésta última va a obtener el contenido del array `popul` para dicho índice, con lo que múltiples hilos pueden tener el mismo valor y generar datos inconsistentes en `cluster_data`. Aprovechamos la directiva `#pragma omp critical` para indicar que se tiene que hacer un **lock** mientras un hilo esté procesando éste fragmento y **unlock** una vez terminado (los demás esperarán). De ésta manera, la ejecución será secuencial para ésta sección de código.

No obstante, existen funciones que se invocan desde otras partes las cuales han sido paralelizadas de manera individual (`nearest_cluster`, `nuevos_centroides`, `silhouette_simple`, `analisis_enfermedades`), como se verá a partir del siguiente apartado.

En cuanto a la muestra de información por pantalla, hemos realizado diversas pruebas y la paralelización no es realmente eficaz, dado que son bucles de tamaño mínimo, y el coste de crear regiones paralelas genera un **overhead**. Como detalle, para solventar el problema del control del clock en regiones paralelas, utilizábamos la directiva `#pragma omp master`, para que fuese ejecutada únicamente por el hilo principal. Por lo general, hemos verificado el siguiente empeoramiento al paralelizar: 0.004 -> 0.07.

Con lo que, hemos decidido que en la versión final - el código de ésta parte estará sin paralelizar.

3.2.2. fun.c

En este fichero hemos paralelizado las funciones del **gengrupos** de manera individual, a excepción de nuevos centroides ya que al utilizar random puede dar resultados impredecibles.

3.2.3. Distancia genética

La función **geneticdist** es una función que calcula la distancia entre dos puntos en un espacio de NCAR = 40 dimensiones, con lo que hemos decidido no paralelizar. Esta función no es adecuada para ser paralelizada debido a que es muy pequeña y no tiene ningún bucle o estructura que merezca ser dividida entre diferentes hilos de ejecución.

Además, es probable que el tiempo necesario para la creación y gestión de los hilos sea mayor que el tiempo necesario para ejecutar la función de manera secuencial. Por lo tanto, la paralelización de esta función **no tendría ningún beneficio**.

3.2.4. Nearest Cluster - grupo más cercano

Código:

```
void nearest_cluster(int nelelem, float elem[][NCAR], float cent[][NCAR], int *samples)
{
    double dist, mindist;
    {
        // Se le invoca desde una región paralela
        // Schedule static por defecto, similar carga para cada iteracion.
#pragma omp parallel for default(none) shared(nelelem, elem, cent, samples, nclusters)\
        private(dist, mindist)
        for(int i = 0; i < nelelem; i++){
            mindist = DBL_MAX;
            for(int k = 0; k < nclusters; k++) {
                dist = geneticdist(elem[i], cent[k]);
                if(dist < mindist){
                    mindist = dist;
                    samples[i] = k;
                }
            }
        }
    }
}
```


Especificación:

Región paralela: `#pragma omp parallel for default(none):`

- `shared(nelem, elem, cent, samples, nclusters)`

Peligro en concurrencia: Las variables `elem`, `cent`, `samples`, `nclusters` son arrays que son accedidos/leídos por los diferentes hilos y algunas dependen de las distintas iteraciones dadas por `i`, `k`, `ncluster`, `nelem`

- `private(dist, mindist)`

Motivo: Necesitamos que cada hilo tenga su propia copia para poder tener control absoluto de las mismas, sin peligro a que otros las modifiquen ya que estamos calculando la distancia mínima para cada una de las muestras, de manera individual.

- **Thread-safe check:** `#pragma omp for` se encarga de repartir el trabajo tal que distintos hilos nunca puedan tener la misma `i`. No es el mismo caso para `k`, pero ésta será privada de manera implícita para cada hilo.

En definitiva, la función `nearest_cluster` paraleliza el procesamiento de cada elemento del array `elem` utilizando la directiva `#pragma omp parallel for` y no hay race conditions en la función, ya que cada hilo trabaja con un elemento del array `elem` y las variables `dist`, `mindist` son privadas.

3.2.5. Silhouette

Código:

```
double silhouette_simple(float samples[][NCAR], struct lista_grupos *cluster_data,
float centroids[][NCAR], float a[]) {

    float b[nclusters];
    float narista = 0;
    double tmp = 0;
#pragma omp parallel default(none)\
        shared(nclusters, b, a, cluster_data, samples, centroids, tmp)\
        firstprivate(narista)
    {
#pragma omp for
        for (int k = 0; k < nclusters; k++) b[k] = 0.0f;
#pragma omp for
        for (int k = 0; k < MAX_GRUPOS; k++) a[k] = 0.0f;

        // tmp puede ser private en lugar de reduction, sin problemas.
#pragma omp for nowait reduction(+: tmp)
        for (int k = 0; k < nclusters; k++) {
            tmp = 0;
            for (int i = 0; i < cluster_data[k].nelems; i++) {
                for (int j = i + 1; j < cluster_data[k].nelems; j++) {
                    tmp += geneticdist(samples[cluster_data[k].elem_index[i]],
                                        samples[cluster_data[k].elem_index[j]]);
                }
            }

            narista = ((float) (cluster_data[k].nelems * (cluster_data[k].nelems - 1)) / 2);
            a[k] = cluster_data[k].nelems <= 1 ? 0 : (float) (tmp / narista);
        }

        // b[k] no depende de nada del bucle anterior, así que quiero que arranque
        // Inmediatamente - nowait.
#pragma omp for nowait
        for (int k = 0; k < nclusters; k++) {
            for (int j = 0; j < nclusters; j++) {
                b[k] += (float) geneticdist(centroids[k], centroids[j]);
            }
            b[k] /= (float) (nclusters - 1);
        }
    }

    // Región ya eliminada
    float max, sil = 0.0f;
    for (int k = 0; k < nclusters; k++) {
        max = a[k] >= b[k] ? a[k] : b[k];
        if (max != 0.0f)
            sil += (b[k] - a[k]) / max;
    }
    return (double)(sil / (float)nclusters);
}
```

Especificación:**Región paralela 1: `#pragma omp parallel for default(none)`:**

- `shared(nclusters, b, a, cluster_data, samples, narista, centroids, tmp)`

Peligro en concurrencia:: Las variables `a`, `b`, `cluster_data`, `samples`, `centroids` son arrays que son accedidos/leídos por los diferentes hilos y algunas dependen de las distintas iteraciones dadas por `i`, `k`, `ncluster`, `nelem`. También, `tmp` está como *shared* para poder ser utilizada con la cláusula *reductions*. Si no, sería *private*.

- `firstprivate(narista)`: variable que inicializamos a 0 fuera de la región paralela pero necesita ser privada para cada hilo.
- `reduction(+: tmp)`

Motivo: Necesitamos que cada hilo tenga su propia variable, pero que al final de la barrera implícita que genera el `#pragma omp for` (no hemos especificado cláusula *nowait*) se actualice la variable `tmp` con la suma de los valores acumulados por cada hilo. (Después de `fork` → `join`)

- **Thread-safe check:** Un caso en cuanto a posibles race conditions, es el de la variable *shared* `tmp`, pero es gestionado automáticamente por OpenMP gracias a la cláusula *reduction*, que hace que cada hilo gestione su valor mediante copia local, para después unificarlo en la misma. Como hemos mencionado, no es necesario en éste caso el uso de *reductions* para evitar race conditions, se podría haber hecho definiendo la variable como privada. Otro que podría aparentarlo, es el del array `a`, que es un array compartido en el cual almacenamos el valor de la densidad para cada clúster. Nuestro primer *reflejo*, fué el de establecer una sección crítica para que no se pudiese modificar su contenido de manera concurrente. Sin embargo, tras analizarlo y al ir asociando los conceptos de paralelización y race conditions, nos hemos percatado de que, como en otras muchas ocasiones, al ser el bucle más externo y sobre el que hemos aplicado la directiva `#pragma omp for`; no existe peligro.

No obstante nos gustaría mencionar a modo de anécdota / proceso de aprendizaje, que una de los grandes bloqueos en ésta función ha sido por culpa de usar en exceso el **nowait**. En los dos primeros bucles que inicializan a 0 los arrays `a`, `b`, pusimos *nowait* muy al principio del proyecto. Nos ha dado grandes dolores de cabeza - una vez teníamos todo funcional (primero en serie; y luego en paralelo trabajando con las 211k muestras), al volver a hacer tests con diferentes combinaciones de hilos y tamaños de muestras, nos hemos encontrado con que para cantidades elevadas de hilos y 1k muestras, teníamos inconsistencias en los resultados de numerosas ejecuciones del programa. Nos costó el darnos cuenta de que la culpa la tenían estos dos 'nowait' de los que 'abusamos' al principio. Pensábamos que eran race conditions.

Región paralela 2: `#pragma omp parallel for default(none)`:

- **Región eliminada:** Hicimos una región paralela extra en éste punto al principio. Ésta función es invocada muchísimas veces y el coste de crear 2 regiones es más alto que el beneficio que se obtiene de la paralelización de un bucle que se va a ejecutar un número bajo de veces (100 como mucho). Hemos dejado nuestra paralelización inicial como comentario a modo de ilustración de nuestra idea inicial.

Como en muchos otros casos (aunque, como hemos explicado → hay que hacerlo analizando el entorno), hemos decidido utilizar la cláusula `nowait` para aquellos bucles que no tienen ninguna dependencia y nos podemos aprovechar de que cada vez que un hilo finalice, se 'salte' la barrera implícita que establece la directiva `#pragma omp for` para que dicho hilo continúe con la siguiente iteración.

En definitiva, la función `silhouette_simple` paraleliza el procesamiento de cada cluster utilizando varias directivas OpenMP. Las variables `nclusters`, `b`, `a`, `cluster_data`, `samples`, `narista`, `centroids` y `tmp` son compartidas entre los hilos y pueden ser leídas o modificadas por cualquiera de ellos. La variable `tmp` es una variable de reducción y se deben acumular los resultados de cada hilo para obtener el resultado final al *traspasar* la barrera implícita.

Por último, el valor de la calidad actual ya no está paralelizado, tras diversos tests hemos contrastado que no merece la pena. Lo que sí habría sido interesante es el crear una **región paralela exterior** a ésta función que invoque ésta y otras, para evitar semejante creación y destrucción de hilos.

3.2.6. Análisis de Enfermedades

```
void analisis_enfermedades(struct lista_grupos *cluster_data, float enf[][TENF],
                          struct analisis *analysis)
{
    int cluster_size, j;
    float median;
    float *disease_data;

    #pragma omp parallel default(none) shared (analysis, nclusters, cluster_data, enf)
    {
        #pragma omp for nowait
        for(int i = 0; i < TENF; i++){
            analysis[i].mmin = FLT_MAX;
            analysis[i].mmax = 0.0f;
        }

        #pragma omp for nowait private(cluster_size, disease_data, median, j)
        for(int k = 0; k < nclusters; k++){
            cluster_size = cluster_data[k].nelems;

            for(j = 0; j < TENF; j++){
                disease_data = malloc(sizeof(float) * cluster_size);

                for(int i = 0; i < cluster_size; i++) disease_data[i] = 0.0f;

                for(int i = 0; i < cluster_size; i++)
                    disease_data[i] = enf[cluster_data[k].elem_index[i]][j];

                median = sort_and_median(cluster_size, disease_data);
            }
        }
    }
}
```

```

        if ((median > 0 && median < analysis[j].mmin) ||
            ((median == analysis[j].mmin) && (k < analysis[j].gmin))){
            analysis[j].mmin = median;
            analysis[j].gmin = k;
        }

        if ((median > analysis[j].mmax) ||
            ((median == analysis[j].mmax) && (k < analysis[j].gmax))){
            analysis[j].mmax = median;
            analysis[j].gmax = k;
        }
        free(disease_data); // Para cada enfermedad
    }
}
}
}

```

Especificación:

Región paralela 1: `#pragma omp parallel for default(none)`:

- `shared (analysis, nclusters, cluster_data, enf)`

Peligro en concurrencia: Las variables `analysis`, `cluster_data`, `enf` son arrays que son accedidos/leídos por los diferentes hilos y algunas dependen de las distintas iteraciones dadas por `k`, `j`.

- `#pragma omp for nowait private(cluster_size, disease_data, median)`:
Dentro del bucle principal, establecemos como privadas estas variables con objeto de que cada hilo tenga su propia copia local y no interfieran los cálculos del resto de hilos.
- **Thread-safe check:** `#pragma omp for`, una vez más, se encargará de realizar el reparto de trabajo e iteraciones tal que distintos hilos no puedan ejecutar el mismo valor de `k`. Con esto, se solventa toda posible *race conditions*. No obstante, hay una parte que nos trajo unos pocos quebraderos de cabeza, la parte del código después de la obtención del valor de `median` con la función `sort_and_median()`. En un principio lo establecimos como `#pragma omp critical`, por los mismos motivos que los explicados en la función anterior para el array `a`. Aquí entra en juego un nuevo índice para el acceso concurrente a distintos arrays, que es el `j`. Sin embargo, al ser un índice también privado para cada hilo, no van a existir *race conditions* y podemos dejarlo tal cual está y ahorrarnos el *overhead* que nos produce la directiva `critical`.

En ésta función hemos utilizado `schedule(dynamic)`. Consideramos que no es estrictamente necesario dados los valores que los clústers pueden tomar, pero como hemos comentado previamente queríamos explorar más directivas y funciones de OpenMP. Entonces, como no podemos predecir el valor de algunos tamaños de clústeres según se está calculando el hyperparámetro `k`, hemos querido introducir éste tipo de *scheduling* para que cada hilo solicite un **nuevo bloque de trabajo** cuando termine de procesar el anterior (aunque sería mejor con números mucho más grandes).

Con lo que, se puede decir que nuestra elección se debe a que el tiempo de ejecución de cada iteración del bucle puede variar significativamente debido a la asignación de memoria dinámica en `disease_data = malloc(sizeof(float) * cluster_size);` y al cálculo de la mediana de los datos de enfermedad, que irá en función del tamaño del clúster. Con `schedule(dynamic)`, cada hilo puede solicitar y recibir más trabajo a medida que termina, lo que puede mejorar el rendimiento en comparación con una asignación estática de bloques de trabajo.

La función `analisis_enfermedades` paraleliza el procesamiento de cada cluster utilizando varias directivas OpenMP. Las variables `analysis`, `nclusters`, `cluster_data` y `enf` son compartidas entre los hilos y pueden ser leídas o modificadas por cualquiera de ellos. Se paraleliza el cálculo de la mediana de las probabilidades de enfermedad para cada clúster y se actualiza la estructura `analysis` de manera segura utilizando la directiva `pragma omp critical`.

3.2.7. Inicializar centroides

Código:

```
void inicializar_centroides(float cent[][NCAR]){
    int i, j;
    srand (147);
    for (i=0; i < nclusters; i++){
        for (j=0; j<NCAR/2; j++){
            cent[i][j] = (rand() % 10000) / 100.0;
            cent[i][j+(NCAR/2)] = cent[i][j];
        }
    }
}
```

Ésta función no la hemos paralelizado. En general, es muy difícil paralelizar funciones que utilizan números aleatorios debido a la naturaleza no determinista de los mismos. Esto se debe a que son generados por una semilla o *seed* (en este caso, 147) y el mismo conjunto de números aleatorios puede ser generado varias veces mediante el uso de la misma semilla. Al utilizar OpenMP para paralelizar, es posible que cada hilo genere un conjunto distinto, lo que puede conducir a resultados impredecibles y posibles problemas de sincronización.

3.2.8. Nuevos Centroides

Código:

```

int nuevos_centroides(float elem[][NCAR], float cent[][NCAR], int samples[], int nelem){

    int i, j, fin;
    double discent;
    double additions[nclusters][NCAR + 1];
    float newcent[nclusters][NCAR];

#pragma omp parallel default(none)\
                shared(nclusters, nelem, samples, elem, additions, fin, newcent, cent)\
                private(i, j, discent)
    {
#pragma omp for
        for (i = 0; i < nclusters; i++)
            for (j = 0; j < NCAR + 1; j++)
                additions[i][j] = 0.0;

#pragma omp for nowait reduction(+: additions[:nclusters][:NCAR+1])
        for (i = 0; i < nelem; i++) {
            for (j = 0; j < NCAR; j++)
                additions[samples[i]][j] += elem[i][j];
            additions[samples[i]][NCAR]++;
        }

        // Que esperen todos los hilos, necesitamos 'additions' correctamente actualizado.
#pragma omp single
        fin = 1;

#pragma omp for
        for (i = 0; i < nclusters; i++) {
            if (additions[i][NCAR] > 0) { // ese grupo (cluster) no esta vacio
                // media de cada caracteristica
                for (j = 0; j < NCAR; j++)
                    newcent[i][j] = (float) (additions[i][j] / additions[i][NCAR]);

                // decidir si el proceso ha finalizado
                discent = geneticdist(&newcent[i][0], &cent[i][0]);
                if (discent > DELTA1)
                    fin = 0; // en alguna centroide hay cambios; continuar

                // copiar los nuevos centroides
                for (j = 0; j < NCAR; j++)
                    cent[i][j] = newcent[i][j];
            }
        }
    }
    return fin;
}

```

Especificación:**Región paralela 1: `#pragma omp parallel for default(none)`:**

- `shared(ncclusters, nelem, samples, elem, additions, fin, newcent, cent)`

Peligro en concurrencia:: Las variables `samples`, `elem`, `additions`, `newcent` y `cent` son arrays que son accedidos/leídos por los diferentes hilos y algunas dependen de las distintas iteraciones dadas por `i`, `j`.

- `reduction(+: additions[:ncclusters][:NCAR+1]):`

Motivo: Establecemos que se reducirá una variable sobre la cual se creará una copia para cada hilo, y una vez todos hayan terminado, los resultados de dichas copias se acumularán en la posición del array `additions` que sea oportuno. Como nota, la sintaxis a especificar para *reduction* en arrays, puede ser como la indicada, donde `[startIndex:endIndex]` para cada dimensión, al no indicar el `startIndex`, se asume es 0.

- `private(i, j, discent)`

Motivo: Como en la función inicial proporcionada los índices `i`, `j` estaban declarados fuera de cualquier bucle, hemos decidido dejarlas así y especificarlas como privadas en la región paralela. La variable `discent` ha de ser creada localmente para cada uno de los hilos.

- **Thread-safe check:** Para que no existan race conditions en la paralelización de ésta función, hemos aplicado los siguientes tratamientos: La ya explicada cláusula `reduction` para el array `additions`, así como el uso de la cláusula con barrera implícita `#pragma omp single`, la cual va a *forzar* que los hilos de ejecución que lleguen a esa línea de código, esperen al resto. Una vez todos hayan llegado a ese punto, entonces se *levantará la barrera* y podrán seguir con la ejecución del resto de código de manera segura. Esto es, para **asegurarnos** de que el array `additions` ha sido actualizado correctamente. Ya que hay una **dependencia directa** en el siguiente loop `for`.

Para esta función, al principio creamos 2 regiones paralelas que excluyesen a `fin`, ya que un hilo puede ir más avanzado que otro y entrar en el siguiente bucle donde el array `additions` aún no está completamente actualizado, al haber introducido la cláusula `nowait`. Pero claro, para eso podemos forzar que sea ejecutado por únicamente un thread (cualquiera(`single`) o máster) ya que poseen una **barrera implícita** y van a provocar que todos los hilos esperen y, una vez todos hayan terminado, pueden proseguir. Con lo que hicimos una sola región dejando el código tal y como se puede apreciar en el fragmento anterior.

En ésta función utilizamos la paralelización para mejorar el rendimiento al realizar los cálculos de los nuevos centroides y la decisión de si el proceso de clustering ha finalizado o no. Al utilizar varios hilos, se pueden realizar estos cálculos de manera más rápida ya que cada hilo puede trabajar en un clúster diferente al mismo tiempo. Además, al utilizar la cláusula `reduction` se asegura que la variable `additions` se modifica de manera segura y que los resultados sean precisos.

4. system.metrics

Para concluir, vamos a someter ambas versiones del programa a una serie de pruebas o tests, durante los cuales mediremos los tiempos de ejecución. Estas pruebas nos permitirán comparar las dos versiones tanto en serie/secuencial como en paralelo, y evaluar el **impacto de la paralelización** del sistema en el rendimiento de la aplicación.

Además, vamos a tener en cuenta distintos factores como el número de muestras tomadas en cada simulación y la cantidad de hilos utilizados, con el fin de obtener una visión más completa del efecto de la paralelización en el rendimiento del sistema.

4.1. Fundamentos teóricos

Antes de comenzar a analizar las métricas, es importante tener una comprensión sólida de los conceptos teóricos y principios fundamentales que nos ayudarán a interpretar y entender mejor los resultados. Estos conceptos pueden incluir leyes, teorías y principios relacionados con el sistema hardware o el área en cuestión, y pueden proporcionarnos un marco teórico sólido para evaluar y analizar los resultados obtenidos.

4.1.1. Incremento de velocidad

Podríamos hacer uso de **Ley de Ahmdahl**, la cual mide el rendimiento y tiempo de ejecución de un proceso o tarea en base a las mejoras que se realicen a una máquina o sistema. Es muy interesante ver cómo se van uniendo ideas - sobre esto hablamos en un trabajo desarrollado el año pasado en la asignatura de Análisis Matemático donde la idea subyacente era el mostrar las distintas aplicaciones de los contenidos de la misma en el mundo de las Ciencias de la Computación, el trabajo puede verse en el siguiente enlace:

[El Análisis Matemático y sus capas de abstracción en las Ciencias de la Computación.](#)

Veamos qué podemos extrapolar y cómo nos podemos nutrir de ello éste proyecto:

$$\text{Incremento Velocidad} = \frac{\text{Rendimiento tras usar mejora}}{\text{Rendimiento SIN mejora}}$$

$$\text{Incremento Velocidad} = \frac{\text{Tiempo de ejecución sin mejora}}{\text{Rendimiento CON mejora}}$$

Veamos la relación con lo visto hasta ahora de la ley de Ahmdahl; Sea T_{sm} el tiempo de ejecución sin mejora y T_{cm} el tiempo de ejecución con mejora, v_{cm} el factor de velocidad-mejora tras la mejora en sí, F_{cm} la fracción de tiempo en el que se utiliza dicha mejora tenemos que:

$$T_{cm} = T_{sm} \cdot \left((1 - F_{cm}) + \frac{F_{cm}}{v_{cm}} \right)$$

Si definimos la mejora de velocidad global v_g como el cociente:

$$v_g = \frac{T_{sm}}{T_{cm}}$$

Podemos usar ésta relación y podemos reformular la ley de Ahmdahl:

$$v_g = \frac{T_{sm}}{T_{cm}} = \frac{1}{1 - F_{cm} + \frac{F_{cm}}{v_m}}$$

Y, por fin, podemos ver la relación con lo estudiado. Si consideramos qué ocurre a la velocidad-mejora global cuando el factor de velocidad-mejora tras la mejora en sí es infinito, obtenemos que:

$$\lim_{v_{cm} \rightarrow \infty} v_g = \frac{1}{1 - F_{cm}}$$

No obstante, para nuestro análisis vamos a simplificar los cálculos y centrarnos el factor de aceleración y eficiencia.

4.1.2. Time-slicing: más threads por núcleo

Un detalle muy importante que, hemos querido investigar ya que no terminábamos de comprender del todo es la *conexión* entre los núcleos de un procesador y el número de hilos que se puede ejecutar.

Ahora podemos decir que es gracias al **Time-slicing**; una técnica utilizada en sistemas de tiempo compartido en los que varios procesos o hilos compiten por el uso del procesador. Esta técnica se basa en dividir el tiempo de procesamiento disponible en unidades más pequeñas; **ticks**, y asignar cada tick a un proceso o hilo en particular para que pueda ejecutarse. De esta manera, cada proceso o hilo recibe una parte del tiempo de procesamiento disponible, lo que le permite ejecutarse de manera intercalada con otros procesos o hilos.

En el caso de los hilos múltiples, el time-slicing se utiliza para **permitir que un núcleo ejecute más de un hilo al mismo tiempo**. Aunque esto puede llegar a reducir el rendimiento del sistema, debido a la sobrecarga adicional que implica dividir el tiempo de procesamiento entre más hilos.

Clúster: 32 núcleos - 64 threads

Sabemos que un procesador puede tener varios cores o núcleos, cada uno de los cuales puede ejecutar un hilo de instrucciones de manera independiente. En el caso de un procesador con 32 núcleos, cada núcleo puede ejecutar un hilo, lo que significa que el procesador puede manejar hasta 32 hilos de manera simultánea.

Sin embargo, gracias al **Time-slicing** es posible que un procesador con varios núcleos pueda utilizar hasta el doble de hilos por núcleo, ya que ésta tecnología permite a un núcleo ejecutar más de un hilo al mismo tiempo.

4.1.3. Factor de aceleración y eficiencia

Visto esto, vamos a expresar de manera *algoritmica* el proceso para realizar los cálculos del factor de aceleración y eficiencia que buscamos tras la paralelización del programa:

Cálculo de factor aceleración y eficiencia

1) Calcular el tiempo de ejecución del programa en serie/secuencial (es decir, con un solo hilo).

2) Para cada conjunto de tiempos de ejecución obtenidos con diferentes hilos, calcular el tiempo promedio.

3) Para tiempo de ejecución obtenido en el paso anterior, calcular el factor de aceleración utilizando la fórmula:

Factor de aceleración = tiempo de ejecución secuencial / tiempo en paralelo.

4) Para cada tiempo de ejecución obtenido en el paso anterior, calcular la eficiencia utilizando la fórmula:

Eficiencia = factor de aceleración / número de hilos

4.2. Análisis de tiempos y eficiencia

En este apartado, analizaremos los tiempos y la eficiencia del programa resultante al ejecutarlo tanto en serie como en paralelo en el clúster.

Para realizar este análisis, nos centraremos en varios indicadores clave, como los **tiempos tomados, el número de hilos de ejecución, el factor de aceleración y la eficiencia**. Los tiempos tomados nos permitirán evaluar cuánto tiempo tarda el programa en completarse en cada caso. Los hilos de ejecución nos permitirán entender mejor cómo se distribuye el trabajo y cómo se está utilizando el clúster; para poder observar cómo afecta esto al rendimiento. El factor de aceleración nos permitirá comparar el rendimiento en paralelo con el rendimiento en serie y ver cuánto se ha acelerado el procesamiento al paralelizar el programa. Por último, la eficiencia nos permitirá evaluar la cantidad de trabajo útil realizado en paralelo en contraposición de la versión en serie.

4.2.1. Best record

Best record

En éste *record*, teníamos la paralelización de la parte de la escritura en consola, lo cual más adelante determinamos que influía negativamente en el rendimiento. Con lo que, es posible el tiempo récord hubiese sido algo menor.

106	Enfermedad: 0 - mmax: 0.41 (grupo 8) - mmin: 0.37 (grupo 5)
107	Enfermedad: 1 - mmax: 0.60 (grupo 3) - mmin: 0.58 (grupo 8)
108	Enfermedad: 2 - mmax: 0.41 (grupo 43) - mmin: 0.38 (grupo 3)
109	Enfermedad: 3 - mmax: 0.61 (grupo 57) - mmin: 0.57 (grupo 8)
110	Enfermedad: 4 - mmax: 0.41 (grupo 5) - mmin: 0.38 (grupo 30)
111	Enfermedad: 5 - mmax: 0.61 (grupo 3) - mmin: 0.58 (grupo 0)
112	Enfermedad: 6 - mmax: 0.41 (grupo 14) - mmin: 0.38 (grupo 5)
113	Enfermedad: 7 - mmax: 0.61 (grupo 3) - mmin: 0.58 (grupo 39)
114	Enfermedad: 8 - mmax: 0.41 (grupo 3) - mmin: 0.38 (grupo 8)
115	Enfermedad: 9 - mmax: 0.61 (grupo 29) - mmin: 0.58 (grupo 25)
116	Enfermedad: 10 - mmax: 0.41 (grupo 7) - mmin: 0.38 (grupo 1)
117	Enfermedad: 11 - mmax: 0.61 (grupo 12) - mmin: 0.58 (grupo 3)
118	Enfermedad: 12 - mmax: 0.41 (grupo 25) - mmin: 0.38 (grupo 12)
119	Enfermedad: 13 - mmax: 0.61 (grupo 34) - mmin: 0.58 (grupo 35)
120	Enfermedad: 14 - mmax: 0.41 (grupo 7) - mmin: 0.38 (grupo 29)
121	Enfermedad: 15 - mmax: 0.61 (grupo 39) - mmin: 0.58 (grupo 36)
122	Enfermedad: 16 - mmax: 0.41 (grupo 19) - mmin: 0.37 (grupo 3)
123	Enfermedad: 17 - mmax: 0.61 (grupo 37) - mmin: 0.57 (grupo 12)

```

Run: genetics_p
↑
↓ Enfermedad: 0 - max: 0.41 (grupo 8) - min: 0.37 (grupo 5)
↓ Enfermedad: 1 - max: 0.60 (grupo 3) - min: 0.58 (grupo 8)
↓ Enfermedad: 2 - max: 0.41 (grupo 43) - min: 0.38 (grupo 3)
↓ Enfermedad: 3 - max: 0.61 (grupo 57) - min: 0.57 (grupo 8)
↓ Enfermedad: 4 - max: 0.41 (grupo 5) - min: 0.38 (grupo 30)
↓ Enfermedad: 5 - max: 0.61 (grupo 3) - min: 0.58 (grupo 0)
↓ Enfermedad: 6 - max: 0.41 (grupo 14) - min: 0.38 (grupo 5)
↓ Enfermedad: 7 - max: 0.61 (grupo 3) - min: 0.58 (grupo 39)
↓ Enfermedad: 8 - max: 0.41 (grupo 3) - min: 0.38 (grupo 8)
↓ Enfermedad: 9 - max: 0.61 (grupo 29) - min: 0.58 (grupo 25)
↓ Enfermedad: 10 - max: 0.41 (grupo 7) - min: 0.38 (grupo 1)
↓ Enfermedad: 11 - max: 0.61 (grupo 12) - min: 0.58 (grupo 3)
↓ Enfermedad: 12 - max: 0.41 (grupo 25) - min: 0.38 (grupo 12)
↓ Enfermedad: 13 - max: 0.61 (grupo 34) - min: 0.58 (grupo 35)
↓ Enfermedad: 14 - max: 0.41 (grupo 7) - min: 0.38 (grupo 29)
↓ Enfermedad: 15 - max: 0.61 (grupo 39) - min: 0.58 (grupo 36)
↓ Enfermedad: 16 - max: 0.41 (grupo 19) - min: 0.37 (grupo 3)
↓ Enfermedad: 17 - max: 0.61 (grupo 37) - min: 0.57 (grupo 12)

t_lec,3.292160
t_clust,45.896847
t_enf,0.070193
t_escr1,0.427249
Texe,49.686450

Process finished with exit code 0

```

Figura 11: Mejor tiempo conseguido en el clúster: 49.68s

4.2.2. Tablas de Tiempos, factor de aceleración y eficiencia

Tamaño de muestras: 1.000 - 50.000

	Núm. hilos	Tamaño de la muestra: 1000			Tamaño de la muestra: 50 000		
		Tiempo medio - seg	Factor de aceleración (fa)	Eficiencia	Tiempo medio	Factor de aceleración (fa)	Eficiencia
Serie	1	0,148874 s	1,000000	1,000000	22,440152 s	1,000000	1,000000
	2	0,093198 s	1,597395	0,798697	12,418335 s	1,807018	0,903509
	4	0,067710 s	2,198700	0,549675	6,989646 s	3,210485	0,802621
	8	0,059323 s	2,509549	0,313694	4,859457 s	4,617831	0,577229
	16	0,052543 s	2,833375	0,177086	5,016129 s	4,473599	0,279600
	24	0,052537 s	2,833698	0,118071	4,776864 s	4,697674	0,195736
	32	0,057773 s	2,576878	0,080527	4,717859 s	4,756427	0,148638
	48	0,056487 s	2,635544	0,054907	4,058565 s	5,529085	0,115189
	64	0,273604 s	0,544122	0,008502	5,081513 s	4,416038	0,069001

Figura 12: Tiempos en serie y 2-64 hilos para 1k-50k muestras.

Tamaño de muestras: 100.000 - 150.000

	Núm. hilos	Tamaño de la muestra: 100 000			Tamaño de la muestra: 150 000		
		Tiempo medio - seg	Factor de aceleración (fa)	Eficiencia	Tiempo medio	Factor de aceleración (fa)	Eficiencia
Serie	1	123,054806 s	1,000000	1,000000	579,149204 s	1,000000	1,000000
	2	67,363074 s	1,826740	0,913370	311,258264 s	1,860671	0,930335
	4	38,036355 s	3,235189	0,808797	167,829836 s	3,450812	0,862703
	8	23,011962 s	5,347428	0,668428	90,945859 s	6,368066	0,796008
	16	16,502079 s	7,456927	0,466058	55,078662 s	10,514947	0,657184
	24	17,093185 s	7,199057	0,299961	45,869244 s	12,626090	0,526087
	32	16,475482 s	7,468965	0,233405	47,859628 s	12,100997	0,378156
	48	13,878740 s	8,866425	0,184717	41,465496 s	13,967015	0,290979
	64	17,019602 s	7,230181	0,112972	40,790381 s	14,198181	0,221847

Figura 13: Tiempos en serie y 2-64 hilos para 100k-150k muestras.

Tamaño de muestras: 211.000

	Núm. hilos	Tamaño de la muestra: 211 000		
		Tiempo medio - seg	Factor de aceleración (fa)	Eficiencia
Serie	1	583,036496 s	1,000000	1,000000
	2	320,453251 s	1,819412	0,909706
	4	179,797536 s	3,242739	0,810685
	8	100,256704 s	5,815437	0,726930
	16	71,860800 s	8,113415	0,507088
	24	58,349887 s	9,992076	0,416336
	32	55,905194 s	10,429022	0,325907
	48	51,898713 s	11,234122	0,234044
	64	50,410576 s	11,565757	0,180715

Figura 14: Tiempos en serie y 2-64 hilos para 211k muestras.

4.2.3. Graficas de los resultados

Tiempos

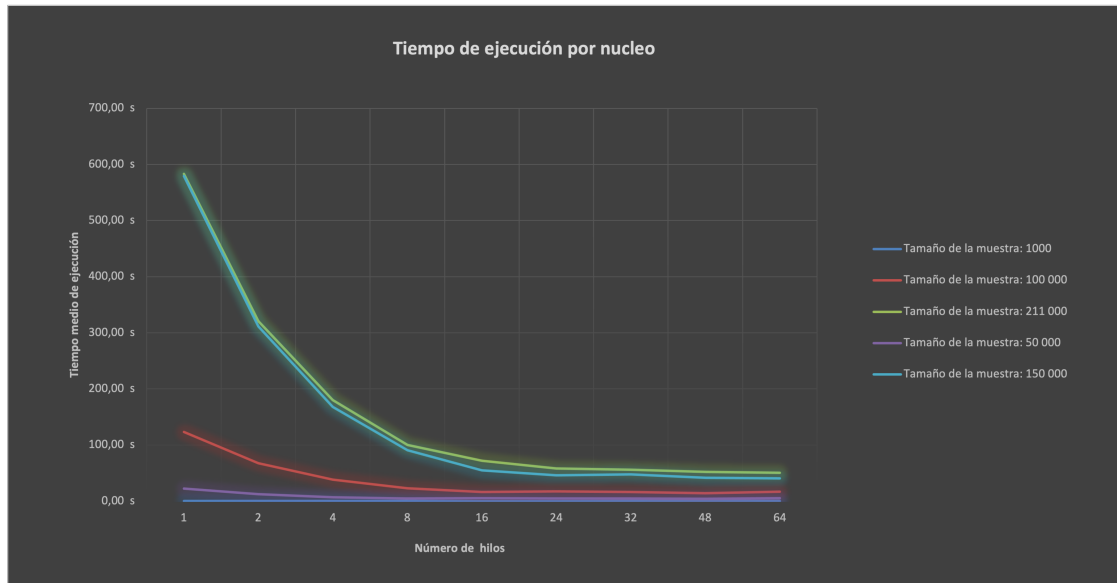


Figura 15: Tiempos de ejecución por núcleo para distintos números de hilos.

Factor de aceleración

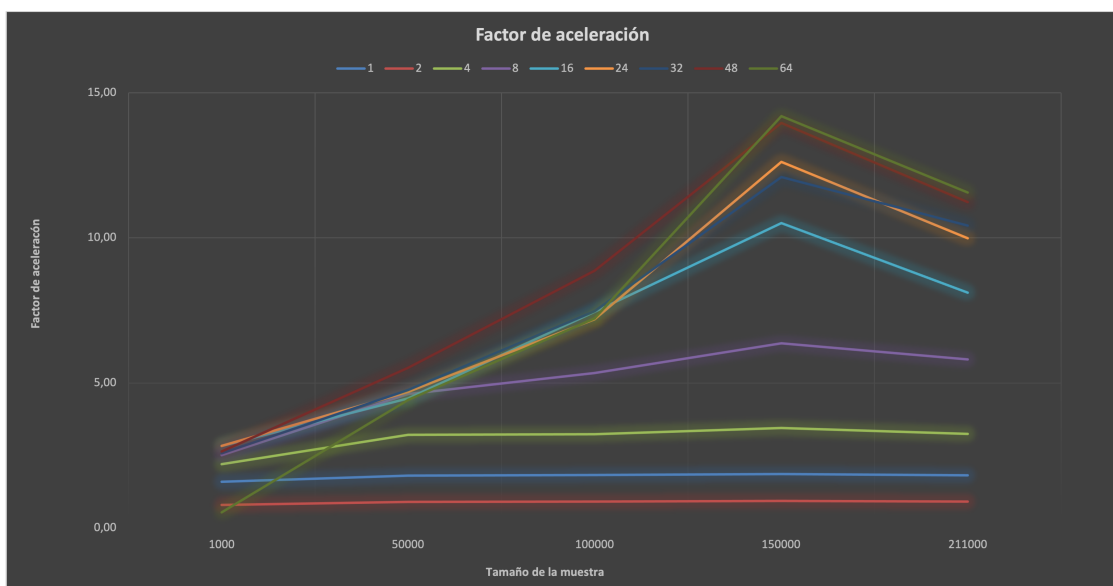


Figura 16: Factor de aceleración para distintos números de hilos.

Eficiencia

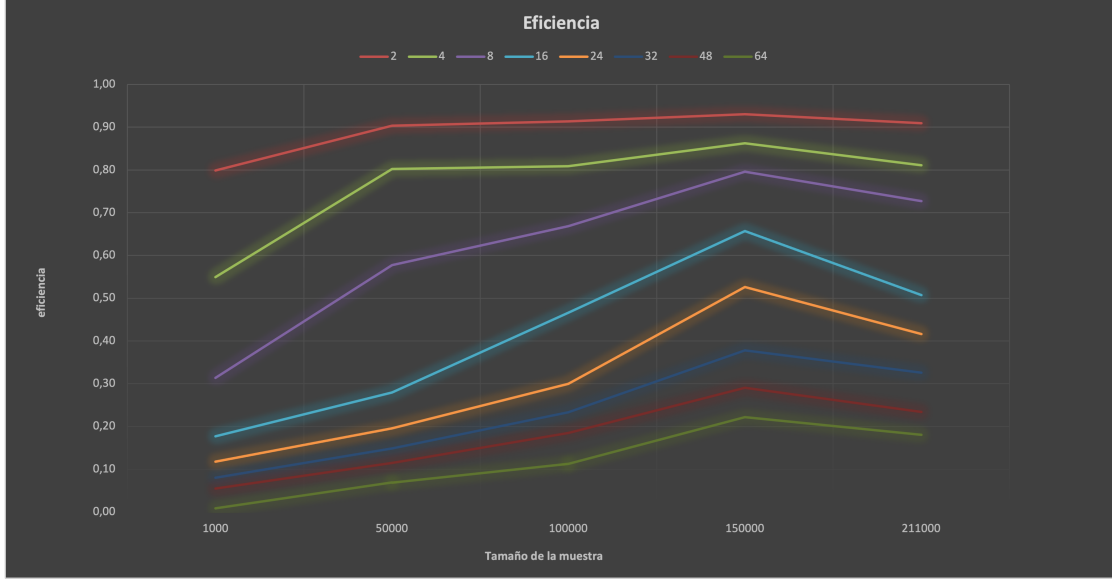


Figura 17: Eficiencia para distintos números de hilos.

4.2.4. Observaciones y conclusiones

En una primera instancia, lo que más nos ha llamado la atención ha sido que, al comparar la **eficiencia con 2 y con 64 hilos**, podemos ver que disminuye significativamente al aumentar el número de hilos - nótese el caso de 1.000 pruebas con 64 hilos, donde obtenemos una eficiencia ínfima de 0.008. Esto puede deberse a que el clúster se está sobrecargando con una cantidad excesiva de hilos de ejecución, lo que lleva a un rendimiento menor en lugar de mejorarlo. Existen diferentes posibilidades; desde un coste extra en términos de **overhead** de sistema, ya que se necesitan más recursos para administrar y coordinar el trabajo de varios hilos, a cuellos de botella o una mala planificación de la paralelización etc. Tras investigar sobre la materia, podemos exponer varios motivos más sobre por qué la utilización de más hilos puede llegar a ser menos eficiente:

- **Sobrecarga de memoria:** Puede haber una sobrecarga de memoria debido a la necesidad de almacenar el contexto y los datos de cada hilo.
- **Contención de recursos:** Si varios hilos están compitiendo por el mismo recurso limitado (como una unidad de disco o una red), esto puede reducir la eficiencia debido a la contención de recursos.
- **Comunicación entre hilos:** Si los hilos deben comunicarse entre sí con frecuencia, esto puede reducir la eficiencia debido al tiempo que se tarda en sincronizar y comunicar los hilos.

En nuestro caso, hemos sido cautos y hemos intentado no abusar de la sincronización (**reductions, critical, barrier/single/master**). Obviamente lo hemos utilizado donde ha sido necesario, pero no creemos que ese sea el problema de eficiencia.

Sigamos con la interpretación de las gráficas.

Otro factor que ha quedado claro es, que a mayor cantidad de hilos y mayor cantidad de muestras, tanto el tiempo de ejecución como factor de aceleración *tienden* a mejorar. Interesante resulta el pico en cuanto al factor de aceleración a partir de 100.000 muestras y con cantidades de hilos altas (16 o más), el cual con el paso de 150.000 a 210.000 vuelva a descender.

Recordemos el hecho de que el clúster - sin **Time-Slicing** - puede ejecutar 32 hilos (1 por núcleo/core). Con lo que, una vez lo ejecutamos con más de 32 el sistema comenzará a hacer uso de este mecanismo y hará que los núcleos que ejecuten más de un hilo hagan que dichos hilos compartan recursos, tal y como nos comentó el profesor; lo cual, en ciertos casos puede no ser beneficioso. Al tener una eficiencia tan baja, implica que hay que tener cuidado ya que no se justifica en algunos casos semejante incremento en la cantidad de hilos.

Como hemos dicho, a mayor sea la cantidad de datos y a mayor sea la cantidad de hilos que los procesan → mayor será el factor de aceleración posee el programa. De esto, se puede apreciar que por el contrario, a una menor cantidad de hilos (2, 4, 8) el programa resulta mucho más eficiente.

Obsérvese, que con el número máximo de muestras - 211k - y 64 hilos, obtenemos el mejor de todos los tiempos medios: 50.41. Sin embargo, estamos **duplicando** la cantidad de hilos con respecto al tiempo medio 55.90, que ha sido logrado con la cantidad de 32 hilos. Si ahora, nos fijamos en el tiempo medio obtenido con 48 hilos, vemos que es mayor que el obtenido 64.

Extraigamos ciertos datos interesantes:

Best of all:

- **Tiempo:** 221k muestras con 64 hilos: 50,41s
- **Factor de aceleración:** 150k muestras con 64 hilos: 14,19
- **Eficiencia:** 150k muestras y 2 hilos: 0,93

Para mejorar la eficiencia de nuestro programa en el clúster, es posible que deba considerar reducir el número de hilos de ejecución para evitar la sobrecarga del sistema y optimizar la distribución de trabajo entre los mismos.

5. Reflexión

Hemos disfrutado muchísimo de la realización de éste proyecto. Para nosotros ha sido un antes y un después. Teníamos ganas de indagar e inmiscuirnos en el desarrollo de aplicaciones con arquitectura paralela y *sabíamos de la existencia* de los threads y libererías que ayudan a utilizarlo. Sin embargo, el comenzar a trabajar con ello y ver lo que se puede llegar a hacer, ha sido todo un descubrimiento - muy motivador.

Hemos observado que cambiando la manera de programar y plantear los problemas, podemos optimizar un sistema en un **1166** %. Impresionante.

Estamos muy agradecidos por el contexto del proyecto - nos ha encantado trabajar con el algoritmo **K-means clustering**. Nos ocurría algo parecedio que con la paralelización - sabíamos que existía pero realmente no conocíamos.

En general, ha sido un proyecto muy enriquecedor que sentimos nos ha hecho crecer.

Por supuesto, no podemos despedirnos sin hacer una referencia al famoso chiste Linuxero de [Don't drink and root](#). Al cual nos gustaría agregar... nor **nowait**!

```
system.shutdown
```

6. Bibliografía

Referencias

- [1] <https://neovim.io>
- [2] <https://www.jetbrains.com/clion/>
- [3] <https://enccs.github.io/cmake-workshop/hello-cmake/>
- [4] <https://www.programiz.com/dsa/quick-sort>

«*Programar es comprender.*»

–Kristen Nygaard

26 de Diciembre de 2022