Universidad del País Vasco Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

PROYECTO DE PARALELIZACIÓN USANDO OPENMP Y C

Análisis genético de muestras de elementos patógenos: Informe técnico

ASIGNATURA: Arquitectura de Computadores
PROFESOR: Iñigo Perona Balda
CURSO ACADÉMICO: 2024/25
GRUPO: G12 (Anne Baquedano, Beñat Descalzo, Maitane Esnaola, Urtzi Peña)

Página intencionalmente dejada en blanco

1. ÍNDICE

2. COMPOSICIÓN DEL GRUPO

- **Código del grupo**: G12
- **Miembros del grupo**:
    - Anne Baquedano Mella
    - Beñat Descalzo Alcuaz
    - Maitane Esnaola Sanchez
    - Urtzi Peña Martin
- **Cuenta**: arq18
- **Directorio**: ~/PROYECTO (ver INDICE.txt para desglose del directorio).

3. DESARROLLO DEL PROGRAMA EN SERIE

El programa que hemos desarrollado realiza un análisis sobre una base de datos con muestras genéticas, y otra con probabilidades asociadas a cada muestra de desarrollar diversas enfermedades. Nuestro programa realiza el siguiente procedimiento:

- **Proceso de clustering sobre las muestras genéticas**: Cada muestra genética está compuesta por 40 "características" (valores que codifican el material genético de la muestra). La cercanía entre dos muestras, matemáticamente, se determina como la distancia euclídea entre los vectores conformados por sus características. Nuestro programa implementa una versión del algoritmo **K-Means clustering**: todas las muestras se agrupan en "clústeres", de tal manera que las muestras en un mismo clúster sean relativamente cercanas, y dos muestras de diferentes clústeres sean relativamente lejanas.
- **Análisis de enfermedades**: Queremos responder a la pregunta "¿Qué grupos son más o menos propensos a desarrollar la enfermedad *n*?". Para ello, el programa analiza todas las enfermedades y grupos, tomando el siguiente criterio:
    - En un grupo, conocemos la probabilidad de que cada muestra desarrolle la enfermedad **n**. Así, la probabilidad de que en dicho grupo se desarrolle la enfermedad **n** es la mediana (el valor en el medio tras ordenar todas las probabilidades).
    - Para cada enfermedad, guardamos qué grupos tienen la mayor y menor probabilidad de desarrollar la enfermedad **n**.

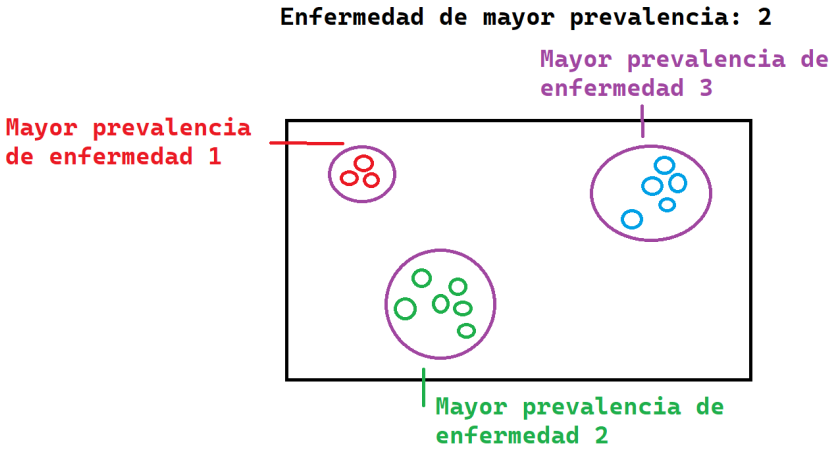


Figura 1: Ejemplo visual de clustering. Los pacientes genéticamente similares se agrupan en 3 "clústeres", y en cada uno predomina una enfermedad. La enfermedad que más prevalece en un grupo es la nº 2.

A continuación, vamos a explicar el diseño del código que implementa estas funcionalidades. Por cuestiones de espacio y claridad, el código está disponible en el servidor interno de la Facultad de Informática, en la cuenta **arq18@dif-cluster.ehu.es**.

Alternativamente, de no funcionar la máquina, todo el contenido se encuentra en la entrega de eGela de este proyecto.

Todos los directorios están marcados en `Monospace`.

3.1 DECISIONES QUE SE HAN TOMADO Y LA DESCRIPCIÓN DEL CÓDIGO

3.1.1 GENDIST

Directorio: `~/PROYECTO/codigo/serie/fun.c`

Esta función es simple: toma dos vectores, elem1 y elem2, que representan dos muestras genéticas de *NCAR* valores. La distancia euclídea se calcula en el bucle for: para cada elemento i de los vectores, calcula su diferencia, la eleva al cuadrado, y la suma al resultado final.
Finalmente, se calcula la raíz cuadrada del resultado. Este diseño responde a la fórmula de la distancia euclídea:

$$d(elem1,\ elem2)\ =\ \sqrt{\sum_{1}^{NCAR}\ (elem1_{i}\ -\ elem2_{i})^{2}})$$

3.1.2 GRUPO_CERCANO

Directorio: `~/PROYECTO/codigo/serie/fun.c`

La función grupo_cercano toma los siguientes parámetros:

- **nelem**: El número de muestras.
- **elem[][NCAR]**: La matriz que contiene todas las muestras, con *NCAR* valores cada una.
- **cent[][NCAR]**: Matriz de centroides. Esta matriz contiene información de cada clúster (*NGRUPOS* clústeres): concretamente, su centroide, o "valor identificativo" (la media de sus elementos). Como el clúster contiene muestras de 40 elementos, el centroide también tiene 40 valores (a cada valor le corresponde la media de dicho valor entre todas las muestras del clúster).
- ***popul**: El array en el que se quiere guardar el resultado.

El trabajo de esta función es calcular el grupo más cercano a cada muestra, guardándolo en el array popul: popul[i]=j si el grupo más cercano al elemento i es el grupo j.

Para ello, se utilizan dos bucles for anidados:

- Para cada elemento i:
    - Para cada grupo j:
        - Calcular la distancia entre el elemento i y el grupo j, usando gendist.
        - Si esta distancia es la primera que se calcula, o mejora la distancia anterior, actualizar la elección de grupo.
    - Se actualiza el vector popul, asignando al elemento i el grupo más cercano que se haya encontrado.

De esta forma, al finalizar la ejecución del for principal, se habrá asignado un nuevo grupo más cercano a todos los elementos.

Nota: La primera asignación aleatoria de clústeres se hace en el fichero gengrupos_s.c, que se nos ha proporcionado de antemano. El código en este fichero ejecuta repetidas veces las funciones de fun.c, hasta que considera que la calidad de los clústeres generados se encuentra dentro del rango aceptable.

3.1.3 SILHOUETTE_SIMPLE

Directorio: `~/PROYECTO/codigo/serie/fun.c`

La función silhouette_simple evalúa la calidad de las particiones, utilizando una versión simplificada del índice de Silhouette. En esta función, s[k] es la calidad de cada clúster k, y S la calidad global del clustering:

- Distancia intra-clúster de cada clúster k:

$$a(k) = \begin{cases} \frac{1}{|c_k|^2} \sum_{x_i \in c_k} \sum_{x_j \in c_k} d(x_i, x_j) & |c_k| > 1, \\ 0 & |c_k| \leq 1. \end{cases}$$

- Distancia inter-clúster de cada clúster k:

$$b(k) = \frac{1}{|C| - 1} \sum_{c_p \in C} d(c_k, c_p)$$

- Calidad de cada clúster k:

$$s(k) = \frac{b(k) - a(k)}{max\{a(k), b(k)\}}$$

- Calidad global del particionamiento:

$$S = \frac{1}{|C|} \sum_{c_k \in C} s(k)$$

Primeramente, se definen como arrays aK[ngrupos], bK[ngrupos] y sK[ngrupos], que van a almacenar el término correspondiente a cada grupo.

Nuestro código para esta función, en su primer apartado, se divide en tres partes, que se ejecutan dentro de un for una vez por grupo:

- **Cálculo del término a[k]**: Con dos bucles for anidados, se suman todas las distancias acorde a la fórmula, y se suman al resultado. Luego, el resultado se guarda como esa suma dividida por la potencia al cuadrado del número de elementos en el grupo k.
- **Cálculo del término b[k]**: En un bucle for, para todos los grupos, se suma al resultado la distancia euclídea del grupo k a cada grupo. Finalmente, se almacena como resultado el valor de esta suma dividido por él (número de grupos - 1).
- **Cálculo del término s[k]**: Se calcula el máximo entre a[k] y b[k] y se aplica la fórmula correspondiente.

Finalmente, dentro de otro bucle for, se calcula el valor S del particionado completo, sumando dentro del bucle todos los s[k] y dividiendo el valor final entre el número de grupos.

Dicho valor S es el que la función devuelve: la calidad del particionado.

3.1.4 ANÁLISIS_ENFERMEDADES

La función analisis_enfermedades calcula, para cada enfermedad, los grupos en los que es más y menos probable que aparezca (definiendo la probabilidad en un grupo como la mediana de las probabilidades de sus muestras).

Para ello, comienza estableciendo unos valores iniciales en un for: máximo de 0 y mínimo de 1 para cada enfermedad.

Después, entra en 2 bucles for anidados. Por cada enfermedad **i**, pasa por todos los grupos **j**, calculando su correspondiente probabilidad de desarrollar la enfermedad, y actualizando los valores en el vector prob_enf si se encuentra un nuevo mínimo o máximo. Para hacer esto, sigue el siguiente proceso en cada iteración del bucle:

- Declarar un vector **probs**, que va a contener la probabilidad de que cada muestra en el grupo desarrolle la enfermedad. El vector se rellena con un bucle for.
- Ejecutar un bubble sort sobre el vector (con dos bucles for anidados), para ordenar las probabilidades y sacar la mediana del grupo.
- Una vez tenemos la mediana (el elemento **listag[j].nelemg/2** de la lista ordenada), la comparamos con el máximo y el mínimo actuales en la enfermedad **i**.
    - Si la mediana del grupo j es mayor que la almacenada en prob_enf[i].mmax, se actualizan sus elementos mmax y gmax.
    - Si la mediana del grupo j es menor que la almacenada en prob_enf[i].mmin, se actualizan sus elementos mmin y gmin.

De esta forma, al completar el bucle principal, el vector **prob_enf** contendrá, para cada enfermedad, los grupos con la mediana mínima y máxima de dicha enfermedad.

3.2 LA EVALUACIÓN DEL PROGRAMA: TIEMPOS OBTENIDOS CON LA VERSIÓN EN SERIE SECCIÓN POR SECCIÓN

Estos son los tiempos que se han obtenido tras una ejecución en una máquina con las siguientes características:

- 2 procesadores Intel Xeon Gold 6130 (16 núcleos, 32 hilos, 2.1 GHz)
- 32 GB RAM (RDIMM)
- Conjunto de instrucciones AVX512

La ejecución se ha realizado el 25 de diciembre de 2024, a las 15:44 horas. Los resultados se encuentran en `~/PROYECTO/resultados_evaluaciones/serie/resSerie.txt`.

Hilos	T_lec	Tiempo de clustering			T_escri	T_enf	T_exe
		T_Silh	T_Grup_cer	T_tot			
1 (serie)	3.166887	309.804314	276.856431	590.110096	0.001999	41.24277	634.521752

Tabla 1: Tiempos de ejecución del código en serie, sin paralelizar.

Para poner en contexto estos datos, el siguiente paso es analizar el código, buscar qué partes son paralelizables (y de qué manera), y comparar los tiempos de ejecución. Con ese análisis podremos, a su vez, calcular la fracción paralela del programa.

Página intencionalmente dejada en blanco

4. LA VERSIÓN PARALELA DEL PROGRAMA

Una vez lista la versión en serie, cuyos tiempos de ejecución hemos analizado en la tabla anterior, hemos desarrollado la versión en paralelo del mismo código, haciendo uso de la librería OpenMP. En esta sección, detallaremos las decisiones que se han tomado (y sus razones) a la hora de paralelizar el código, así como los resultados obtenidos a la hora de ejecutar el programa con diferente cantidad de hilos. El objetivo de la paralelización, en este caso, es aprovechar la existencia de múltiples hilos para ejecutar un mismo programa en menos tiempo.

4.1 ESTRATEGIAS PARA EL REPARTO DE TRABAJO, Y NECESIDADES DE SINCRONIZACIÓN

4.2.1 MAIN

Dentro de la función main, hemos decidido **no paralelizar la mayoría de los bucles**. Las razones son las siguientes:

- **Bucles de lectura-escritura**: Hemos preferido no paralelizar bloques de código que trabajaran con la entrada/salida por seguridad, para evitar posibles conflictos. De todas formas, aún pudiendo garantizar que no hubiera ningún problema, el tiempo que supone el I/O en el programa es casi imperceptible, como se puede observar en los resultados en serie (p. ej. 0.02s para la escritura), así que el overhead introducido por la paralelización no compensa la diferencia.
- **Bucles for que realizan llamadas a las funciones en fun.c**: Cuando los bucles llaman a funciones como grupo_cercano, no debemos paralelizarlas, para evitar paralelización de varios niveles. Estas funciones ya se encuentran paralelizadas adecuadamente en su implementación.
- **Bucle que calcula el número de elementos en cada grupo**: Al haber >200K iteraciones en este bucle, y no haber ningún conflicto con el I/O o con funciones ya paralelizadas, hemos paralelizado este bucle.

Cabe destacar que este bucle ha causado problemas de sincronización: concretamente, la sección en la que se asigna a num el valor de listag[grupo].nelemg, y se suma uno al valor justo después.

La solución ha sido agrupar estas dos operaciones en una sección crítica del bucle. Esta podría no ser la solución más eficiente, pues a efectos prácticos el bucle se acerca a su versión en serie, pero nos ha permitido poner a prueba el funcionamiento de la cláusula `#pragma omp critical`.

4.2.2 GENDIST

Al paralelizar la función gendist, probando una muestra de mil elementos, nos hemos dado cuenta de un detalle importante: el *overhead* introducido por el proceso de paralelización descompensa el tiempo de ejecución (la función se ralentiza). Por ello, hemos decidido **no paralelizar** gendist y, en su lugar, cuando otra función ejecute varias instancias de gendist (p. ej. silhouette_simple, como veremos más adelante), distribuir esas instancias entre varios núcleos.

Esta ralentización es lógica, pues gendist es una función que en su bucle interno tan solo realiza 40 restas, 40 multiplicaciones, y 40 sumas. Esta operación es virtualmente instantánea en un solo core, por lo que añadir el overhead de distribuir las 40 operaciones es innecesario.

4.2.3 GRUPO_CERCANO

Esta función tiene dos bucles for anidados: para cada muestra, se ejecuta gendist en todos los grupos. En este caso, el grueso de la computación sucede en el bucle interior (que recorre los grupos), por lo que a efectos prácticos, no supone una gran diferencia paralelizar el bucle interno o externo. Sin embargo, con el fin de reducir el overhead, hemos establecido la paralelización en el bucle exterior.

Situar la instrucción `#pragma omp for` en el bucle interior supondría, por cada grupo, volver a iniciar el trabajo de paralelización desde cero, mientras que, en el bucle externo, la única carga es la de mantener el reparto de trabajos.

Respecto al reparto de trabajo: nuestra teoría es que el reparto **estático** es el más adecuado para esta función. Es una función con un tiempo de ejecución estable: en cada iteración del bucle externo, hay *[ngrupos]* ejecuciones de la función gendist(), de tiempo de ejecución constante, por lo que no hay ninguna descompensación de la que se beneficie el reparto dinámico.

4.2.4 SILHOUETTE_SIMPLE

En la función silhouette_simple, en vez de paralelizar el bucle principal (con una iteración por grupo), hemos decidido paralelizar individualmente los distintos bucles internos que lo conforman, con los que se calcula cada término del índice Silhouette. Concretamente, hay dos bucles internos diferentes: uno para el término a[k] y otro para el término b[k]. En ambos, se ejecuta una llamada a la función gendist() una cierta cantidad de veces.

Nuestra teoría es que ambos bucles se benefician de un reparto estático de tareas, pues cada llamada a gendist debería, en teoría, durar lo mismo. Probaremos o rechazaremos estas teorías en los resultados de las evaluaciones.

4.2.5 ANÁLISIS DE ENFERMEDADES

Para paralelizar esta función, teníamos tres opciones:

- **Un hilo por enfermedad** (bucle exterior).
- **Un hilo por grupo al analizar cada enfermedad** (bucle interior)..
- **Paralelizar alguno de los bucles internos**.

La última opción es la más sencilla de descartar: el bucle interno más corto (que asigna los valores de probs[k]) no es el origen de la lentitud de esta función. El otro bucle, que realiza una ordenación en burbuja, **no puede paralelizarse**: cada iteración es dependiente de la anterior.

Ahora bien, ¿por qué hemos escogido paralelizar un hilo por enfermedad frente a un hilo por grupo? Para responder a esta pregunta, tenemos que analizar las ventajas y desventajas de cada elección:

- **Un hilo por enfermedad**: Escogiendo esta opción, el overhead para el proceso de paralelización es pequeño. Esto es positivo si tenemos una cantidad de enfermedades similar al número de núcleos (en este caso, 18 enfermedades).
  El problema surge cuando tenemos una gran cantidad de hilos disponible (por ejemplo, 64), pues se están desaprovechando.
- **Un hilo por grupo**: En este caso, el overhead es mayor, pero tenemos el beneficio de poder poner a trabajar todos los núcleos al mismo tiempo. Sin embargo, si tenemos una cantidad de hilos muy pequeña, tendremos que estar asignando nuevas tareas a los hilos todo el tiempo, sumando una sobrecarga indeseada.

Poniendo ambas a prueba para analizar la validez de la hipótesis, hemos obtenido estos resultados (valores en segundos):

	Bucle exterior (rep. estático)	**Bucle interior (rep. estático)**
2 cores	20.687001	28.966372
64 cores	2.338673	2.083777

Tabla 2: Tiempos de ejecución de la función analisis_enfermedades, según el bucle for que se decida paralelizar.

Como podemos observar, con muchos núcleos la diferencia de tiempo no es significativa, pero, con muy pocos núcleos, nos podemos beneficiar enormemente de paralelizar el bucle exterior (**un hilo por enfermedad)**. Por esta razón, hemos considerado adecuada la primera opción.

Para el reparto de trabajo, creemos que lo mejor es utilizar un reparto estático. En el contexto de ejecución de este programa (equipos de la OMS), se espera tener una gran cantidad de núcleos disponibles. En el caso de los datos de prueba que estamos utilizando, con 18 enfermedades, a cada enfermedad se le puede asignar un único núcleo. Sin embargo, si estuviéramos utilizando un reparto dinámico con más de una iteración en bloque, estaríamos limitando el número de hilos a 9 o menos, perdiendo rendimiento.

4.2.6 INIT_CENT_RAND

Hemos decidido no paralelizar esta función, pues realiza una cantidad pequeña (2*ngrupos*NCAR/2 ~ 720) de operaciones elementales. El overhead introducido por la paralelización no compensa la diferencia.

4.2.7 NEW_CENTROIDES

Hemos decidido no paralelizar esta función, basándonos en resultados experimentales obtenidos. Al ser una función relativamente veloz (200K operaciones de suma, ~80*40 divisiones…), pero que se ejecuta repetidas veces desde el main, el overhead introducido por la paralelización ralentiza más que ayudar.

En 64 núcleos, hemos visto que el cálculo de grupos cercanos pasa de ~7.49s a ~36.74s por el mero hecho de paralelizar la actualización de los centroides.

5. ANÁLISIS DEL RENDIMIENTO

En esta sección, vamos a analizar el rendimiento de nuestro código al ejecutarlo con varios hilos paralelos funcionando simultáneamente. De esta forma, podremos analizar la diferencia de rendimiento respecto a la ejecución en serie del código, y sacar diversas conclusiones respecto al número de hilos, las modalidades de reparto de trabajo, etc.

Todos los resultados se encuentran en `~/PROYECTO/resultados_evaluaciones`.

5.1 METODOLOGÍA

Nuestro código, en la función gengrupos, cronometra los siguientes tiempos:

- Tiempo de lectura (T_lec).
- Tiempo de clustering:
    - Tiempo de cálculo de grupos más cercanos (T_Grup_cer).
    - Tiempo de cómputo del índice Silhouette (T_Silh).
    - Tiempo total de clustering (T_tot, o, en los ficheros de output, T_clust)
        - Incluye otras operaciones menores, que añaden ~10 segundos.
- Tiempo de análisis de enfermedades (T_enf).
- Tiempo de escritura (T_escri).

Tomaremos como **tiempo total** (T_exe) la suma de los tiempos de lectura, clustering, análisis de enfermedades y escritura. El tiempo total de ejecución es unos milisegundos mayor, por operaciones ajenas a estos procesos, pero ese overhead es irrelevante para este análisis.

Hemos realizado las siguientes ejecuciones:

- Ejecución del código en paralelo con 2, 4, 8, 16, 32 y 64 hilos, con **todos los bucles en reparto estático**.
- Ejecución del código en paralelo, con **todos los bucles en reparto dinámico** de 1, 2, 4, 8, 16, 32 y 64 tareas.
    - Para cada cantidad de tareas, hemos realizado la ejecución con todas las cantidades de hilos mencionadas anteriormente.

Gracias a esas ejecuciones, y a la medición atómica de los tiempos, podemos comprobar si nuestras hipótesis del informe se sostienen o no. En la siguiente sección, se presentan tablas con todos los resultados, desglosados por número de tareas, número de hilos y secciones del programa. También se ofrecen algunos cálculos relacionados con el rendimiento de cada sección, útiles para alcanzar conclusiones finales.

5.2 RESULTADOS POR REGIÓN Y POR REPARTO

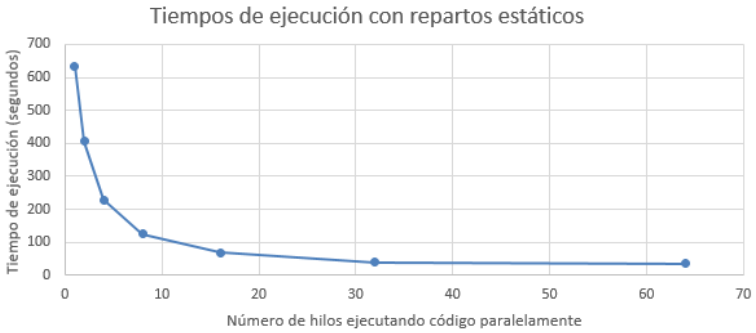
5.2.1 RENDIMIENTO: REPARTO ESTÁTICO

Vamos a comenzar por el **reparto estático** de todos los bucles. Hemos decidido mantener el tamaño de reparto por defecto (iteraciones/hilos). Estos son los resultados de una batería de ejecuciones, a día 25 de diciembre (15:16 horas):

Hilos	T_lec	Tiempo de clustering			T_escri	T_enf	T_exe
		T_Silh	T_Grup_cer	T_tot			
64	3.286157	17.478615	6.773271	30.249342	0.002553	2.329483	35.867535
32	3.284153	20.136832	9.030157	33.231942	0.002155	2.312761	38.831011
16	3.27835	39.240852	17.597038	60.736648	0.002113	4.595674	68.612785
8	3.283851	75.197716	35.390464	114.494569	0.002162	6.917232	124.697814
4	3.285174	139.991439	69.776674	213.651106	0.002085	11.48149	228.419855
2	3.286574	238.961509	139.229036	381.784777	0.00205	20.645832	405.719233
1 (serie)	3.166887	309.804314	276.856431	590.110096	0.001999	41.24277	634.521752

Tabla 3: Tiempos de ejecución con reparto estático de los bucles paralelizados. Todos los tiempos están en segundos.

La opción más sencilla para analizar el comportamiento de este programa es graficar el tiempo total de ejecución con respecto a los hilos:



Speed-ups relevantes

$$fa_{16} = \frac{T_{serie}}{T_{16}} = \frac{634.52}{68.61} = 9.248$$

$$fa_{32} = \frac{T_{serie}}{T_{32}} = \frac{634.52}{38.83} = 16.34$$

$$fa_{64} = \frac{T_{serie}}{T_{64}} = \frac{634.52}{35.86} = 17.69$$

Figura 2: Evolución del tiempo de ejecución en función del número de hilos.

Con estos datos, son inmediatas algunas conclusiones importantes.

Para empezar, la paralelización tiene un gran impacto positivo sobre el rendimiento. Con 64 hilos, hemos obtenido una ejecución casi 18 veces más rápida, con resultados correctos. Con el tamaño de la base de datos que hemos utilizado (200K muestras), la ejecución ha pasado de tardar más de 10 minutos a completarse en 36 segundos. Quizás en este contexto no sea una diferencia tan relevante, pues esa única ejecución de 10 minutos es más rápida que el tiempo invertido en paralelizar, pero esta aceleración puede volverse crucial en un contexto de millones de muestras.

Por otro lado, hemos notado que la ganancia no es lineal: el speed-up, en nuestros datos, tiene una tendencia logarítmica. Dicho de otra forma, tenemos "diminishing returns" en la ejecución. El salto de 16 a 32 hilos ofrece casi el doble de aceleración, pero al pasar de 32 a 64, teniendo el doble de hilos, solo obtenemos un beneficio marginal de unos 3 segundos. Esto nos demuestra que la intuición de "cuantos más hilos mejor" no es necesariamente correcta.

El problema va más allá: en algunos casos, se observa un posible **efecto nocivo** al paralelizar de más. Veamos el caso de t_enf (el tiempo de ejecución de la función analisis_enfermedades). Con 32 núcleos, hemos obtenido un tiempo de 2.312761, mientras que, con 64 núcleos, ese tiempo pasa a 2.329483. ¿Cómo puede suceder esto?

Tenemos que recordar un dato importante: **la sobrecarga es función del número de núcleos/hilos**. Con 64 hilos, la sobrecarga es mayor que con 32, pues hay que analizar un reparto más complejo. En analisis_enfermedades, enviamos, como mucho, 40 tareas, pues hay 40 enfermedades en nuestros datos. Por tanto, al pasar de 32 a 64 hilos, tenemos un efecto doble:

- **Aprovechar 8 hilos más**: Este factor contribuye a acelerar la ejecución, pues no hay ningún hilo que tenga que realizar 2 trabajos seguidos.
- **Coordinar la ejecución paralela teniendo en cuenta los 24 hilos sobrantes**: Aunque lógicamente estos hilos no van a ser usados, decidir qué hacer con ellos es una tarea que la CPU tiene que realizar. Es una tarea casi instantánea, pero que puede ser suficiente para descompensar la diferencia del punto anterior.

NOTA: Estos son los datos de una ejecución a una hora concreta, con una carga concreta en el ordenador en el que se han realizado las operaciones. Con una diferencia tan pequeña (0.016 segundos), hay muchos factores externos a los que se puede achacar la diferencia (p. ej. más usuarios conectados, tareas de mantenimiento que esté ejecutando el sistema de fondo, etc.), pero el concepto se mantiene: **los núcleos de más dejan de ofrecer un beneficio tangible**. En otro par de ejecuciones arbitrarias, podría darse el caso de que la versión paralela tenga esos milisegundos de ventaja, pero es un hecho incierto.

Antes de detallar la eficiencia de las funciones individuales, vamos a hacer un análisis de las funciones que han sido ejecutadas en un régimen dinámico de reparto, para decidir la mejor configuración posible.

5.2.2 RENDIMIENTO: REPARTO DINÁMICO

Para el análisis del reparto dinámico, como ya hemos mencionado, hemos realizado una gran cantidad de pruebas (49 ejecuciones distintas, concretamente). Los resultados "en crudo" de estas ejecuciones se encuentran en el directorio de la máquina remota, y no merece la pena mostrarlos todos en este informe. En su lugar, vamos a presentar algunos de los resultados más relevantes, de los que hemos podido extraer una lectura interesante.

Antes que nada, ¿se cumplen nuestras teorías respecto al funcionamiento del código? Recapitulemos (presentando los resultados en 64 hilos de ejecución):

- **analisis_enfermedades**: Hemos supuesto que el reparto estático sería la mejor opción, para evitar perder hilos con tan pocas iteraciones. Estos son los resultados que hemos obtenido:

Estático	1 tarea	2 tareas	4 tareas	8 tareas	16 tareas	32 tareas	64 tareas
2.329483	2.411232	4.707658	9.325882	18.607449	37.151630	41.747422	41.746248

Tabla 4: Tiempos de ejecución de analisis_enfermedades en 64 hilos, según el número de iteraciones que se reparten dinámicamente.

Podemos ver que nuestra hipótesis se cumple: Repartiendo una tarea por hilo, no hay problema, pero a partir de ahí, aumentar el número de tareas que se reparte a cada hilo implica reducir el número de hilos disponibles.

- **silhouette_simple**: En esta función, nuestra teoría era que la ejecución se beneficiaría de dos repartos estáticos, pues cada iteración dura un tiempo similar, pero hemos pasado por alto un detalle: las iteraciones en el cálculo de a[k] son desiguales. Esto se refleja en los resultados que hemos obtenido:

Estático	1 tarea	2 tareas	4 tareas	8 tareas	16 tareas	32 tareas	64 tareas
17.478615	6.961510	10.204170	7.230631	7.346456	7.957909	9.860203	16.615647

Tabla 5: Tiempos de ejecución de silhouette_simple en 64 hilos, según el número de iteraciones que se reparten dinámicamente.

Hemos realizado dos ejecuciones para comprobar esta teoría (~/PROYECTO/resultados_evaluaciones/evaluaciones_express). En una de ellas, el bucle correspondiente a a[k] tiene reparto dinámico de a 4 tareas, y el bucle b[k] un reparto estático. En la otra, lo contrario: a[k] en reparto estático, y b[k] en reparto dinámico. Hemos elegido 4 tareas por ser, según los resultados anteriores, el valor óptimo (por una razón similar a la de analisis_enfermedades).

	a[k] dinámico, b[k] estático	a[k] estático, b[k] dinámico
Tiempo de silhouette_simple	7.190050	17.053471

Tabla 6: Tiempos de ejecución de silhouette_simple en 64 hilos, según el bucle cuya carga de trabajo decide repartirse de manera dinámica.

- **grupo_cercano**: Para encontrar el grupo más cercano a cada muestra, hemos supuesto que el reparto estático sería adecuado, pues en todas las iteraciones, la parte computacionalmente pesada es la ejecución de **gendist**. No nos hemos equivocado, pero una conclusión más adecuada, según los datos, es que el reparto no supone ninguna diferencia significativa:

Estático	1 tarea	2 tareas	4 tareas	8 tareas	16 tareas	32 tareas	64 tareas
6.773271	7.763772	8.026949	6.802390	6.727688	6.661720	6.995334	6.749355

Tabla 7: Tiempos de ejecución de grupo_cercano en 64 hilos, según el número de iteraciones que se reparten dinámicamente.

La fluctuación aparentemente arbitraria de los valores demuestra que, en el mejor de los casos, el reparto dinámico puede ofrecer una ventaja de unos pocos milisegundos, mientras en el peor, puede sumar más de un segundo a la ejecución. Por tanto, consideramos que el reparto estático es una apuesta segura.

5.2.3 REFLEXIÓN Y DECIDIR LA CONFIGURACIÓN

Tras evaluar los tiempos de ejecución de las diferentes funciones, hemos decidido que esta es la configuración ideal para ejecutar el programa:

- **Número de hilos**: 64 hilos. Todas las funciones se benefician del máximo número de hilos disponible (algunas más, como la de clustering).
- **Main**: El bucle paralelizado va a tener un reparto estático, pues sus iteraciones son idénticas.
- **Analisis_enfermedades**: Esta función se va a ejecutar con un reparto estático.
- **Silhouette_simple**: El primer bucle for paralelizado (correspondiente a a[k]) se va a ejecutar con un reparto dinámico, con asignación de a 4 iteraciones. El segundo bucle (correspondiente a b[k]) se va a ejecutar con un reparto estático.
- **Grupo_cercano**: Esta función se va a ejecutar con un reparto estático.

El código con esta configuración se encuentra en `~/PROYECTO/codigo/paralelo_optim`. En la siguiente sección, vamos a ejecutar el código con esta configuración, para calcular el speed-up, la eficiencia de las diferentes funciones, y diferentes métricas interesantes sobre el rendimiento..

5.3 SISTEMA ENTERO: RENDIMIENTO

Veamos los resultados obtenidos al ejecutar el código con la configuración óptima. Las ejecuciones paralelas se han realizado el 26 de diciembre de 2024, a las 14:37h:

Hilos	T_lec	Tiempo de clustering			T_escri	T_enf	T_exe
		T_Silh	T_Grup_cer	T_tot			
64	3.28768	7.146994	6.771938	18.723506	0.014196	2.32297	24.348352
32	3.292809	9.992137	9.86759	23.592794	0.00214	2.360693	29.248436
16	3.290253	19.495215	17.645381	40.723185	0.002105	4.637178	48.652721
8	3.286061	38.927404	35.139371	77.620120	0.002065	6.896205	87.804451
1 (serie)	3.166887	309.804314	276.856431	590.110096	0.001999	41.24277	634.521752

Tabla 8: Tiempos de ejecución en segundos del programa completo, con las configuraciones de reparto "óptimas" para cada bucle.

De aquí podemos extraer varias métricas, tanto globales como para cada función:

5.3.1 MÉTRICAS GLOBALES

Empecemos viendo el speed-up y eficiencia que obtenemos según la cantidad de hilos:

Hilos	**Speed-up** ($\frac{Tiempo\ serie}{Tiempo\ en\ paralelo}$)	**Eficiencia** ($\frac{Speed-up}{Nº\ de\ núcleos}$)
64	26.0601519	0.407189873 ~ 40.7%
32	21.6942113	0.677944103 ~ 67.7%
16	13.0418554	0.815115962 ~ 81.5%
8	7.22653288	0.90331661 ~ 90.3%

Tabla 9: Speed-up (factor de aceleración) y eficiencia obtenidas en la ejecución paralelizada del programa, según el número de hilos utilizado.

Podemos usar la **Ley de Amdahl** para estimar la fracción de código que hemos ejecutado en paralelo. Por ejemplo, en 64 y 32 hilos:

$fa = \frac{P}{f+(1-f)P}$ (**Ley de Amdahl**, sobre el límite teórico del speed-up)

$$fa_{64} = 26.06 = \frac{64}{f+(1-f)*64} = \frac{64}{f+64-64f} = \frac{64}{64-63f};\ 26.06 * (64 - 63f) = 64;\ f \approx 97\%$$

$$fa_{32} = 21.69 = \frac{32}{f+(1-f)*32} = \frac{32}{f+32-32f} = \frac{32}{32-31f};\ 21.69 * (32 - 31f) = 32;\ f \approx 98\%$$

No podemos obtener una cifra exacta, pues los tiempos de ejecución reales están influenciados por factores externos al propio código. Sin embargo, podemos observar que el código es altamente paralelizable: una fracción paralela cercana al 98% del código ejecutado.

5.3.2 MÉTRICAS POR FUNCIÓN

Ahora, podemos calcular los speed-ups y eficiencias de las 3 funciones principales.

- **Analisis_enfermedades**: Observamos que, a partir de los 32 núcleos (tal y como hemos explicado anteriormente), esta función es la que menos se beneficia de una mayor paralelización (con una gran pérdida de eficiencia, en consecuencia).

Hilos	**Speed-up** ($\frac{Tiempo\ serie}{Tiempo\ en\ paralelo}$)	**Eficiencia** ($\frac{Speed-up}{Nº\ de\ núcleos}$)
64	17.7543274	0.277411366 ~ 27.7%
32	17.4706199	0.545956872 ~ 54.6%
16	8.89393722	0.555871076 ~ 55.6%
8	5.98050232	0.74756279 ~ 74.8%

Tabla 10: Speed-up y eficiencia obtenidas en la ejecución paralelizada de analisis_enfermedades.

- **Silhouette_simple**: Esta es la función que más se ha beneficiado de la paralelización. Con un speed-up máximo de 43.347x en 64 hilos, muestra una eficiencia más que razonable del 67.7%.

Hilos	**Speed-up** ($\frac{Tiempo\ serie}{Tiempo\ en\ paralelo}$)	**Eficiencia** ($\frac{Speed-up}{Nº\ de\ núcleos}$)
64	43.3474988	0.67730467 ~ 67.7%
32	31.0048105	0.96890033 ~ 96.9%
16	15.8913002	0.99320626 ~ 99.3%
8	7.95851462	0.99481433 ~ 99.5%

Tabla 11: Speed-up y eficiencia obtenidas en la ejecución paralelizada de silhouette.

- **Grupo_cercano**: Si bien no se ha obtenido el mismo resultado que en silhouette_simple, la paralelización en grupo_cercano también ha ofrecido resultados excelentes, además de escalables con una mayor cantidad de hilos (sin una pérdida tan drástica de eficiencia).

Hilos	**Speed-up** ($\frac{Tiempo\ serie}{Tiempo\ en\ paralelo}$)	**Eficiencia** ($\frac{Speed-up}{Nº\ de\ núcleos}$)
64	40.8828951	0.63879524 ~ 63.9%
32	28.0571478	0.87678587 ~ 87.7%
16	15.6900228	0.98062643 ~ 98.1%
8	7.8788101	0.98485126 ~ 98.5%

Tabla 12: Speed-up y eficiencia obtenidas en la ejecución paralelizada de grupo_cercano.

Página intencionalmente dejada en blanco

6. CONCLUSIONES

Concluimos este reporte con algunas conclusiones de los datos que hemos analizado en el anterior punto.

Para empezar, la pregunta más evidente: ¿Merece la pena paralelizar? La respuesta es un rotundo **sí**. Aplicando un gran número de hilos (64), hemos conseguido un tiempo de ejecución 26 veces más pequeño, pero ni siquiera es necesario un procesador tan complejo para obtener resultados sorprendentes. Usando tan solo 4 hilos, logramos reducir la ejecución en más de la mitad del tiempo.

En este programa, la sección que más ralentiza la ejecución en serie era la generación de clústeres, entre el cálculo del grupo más cercano a cada elemento y el índice de Silhouette. Esta sección ha sido, a su vez, la más paralelizable de todas, logrando un speed-up de 31.517x. Cabe destacar que, de sus funciones, la más paralelizable ha sido el índice de Silhouette, con un speed-up de 43.347x entre todas sus ejecuciones.

Sin embargo, surge una cuestión importante: ¿Cuán eficiente es este uso de los recursos? Está claro que, si se desea el mejor tiempo sin importar los recursos, la mejor opción (en este procesador) es el uso de 64 hilos. Ahora bien, esto implica llevar al procesador a su límite (que quizás es lo que se busca, emplear toda su potencia en una única tarea).

Esta opción, como hemos comprobado en los cálculos de eficiencia, no es la que mejor aprovecha los recursos a su alcance: con 32 hilos, se obtienen resultados de tiempo muy similares usando la mitad de los recursos. Con 16 hilos tampoco se obtienen malos resultados, una ejecución 13 veces más veloz del programa original. Para reducir este tiempo a la mitad, habría que cuadruplicar el número de hilos a 64. Es decir: en este programa, si se persigue equilibrar la eficiencia con el tiempo de ejecución, lo más adecuado es usar 16 o 32 hilos, según cuánto se valore el tiempo ahorrado con 32 hilos.

En resumen, la paralelización es una técnica con un gran potencial para optimizar la ejecución de aplicaciones grandes, pero tiene sus limitaciones, y debe usarse con cabeza para aprovechar los recursos de la manera más eficiente posible (y evitar ralentizar más el código).

7. BIBLIOGRAFÍA

Esta lista contiene algunos de los principales recursos que hemos utilizado para elaborar este trabajo.

- Apuntes de paralelización y OpenMP en la plataforma eGela (UPV/EHU, 2024).
- Guía de referencia de OpenMP (OpenMP, 2024): https://www.openmp.org/wp-content/uploads/OpenMPRefGuide-5.2-Web-2024.pdf
- *"Environment variables for OpenMP"*, documentación no oficial de IBM (IBM, 2021): https://www.ibm.com/docs/en/xffbg/121.141?topic=options-environment-variables-openmp
- *"Chapter 3: Multithreading with OpenMP",* Programming Parallel Computers (Universidad de Aalto, 2015): https://ppc.cs.aalto.fi/ch3/