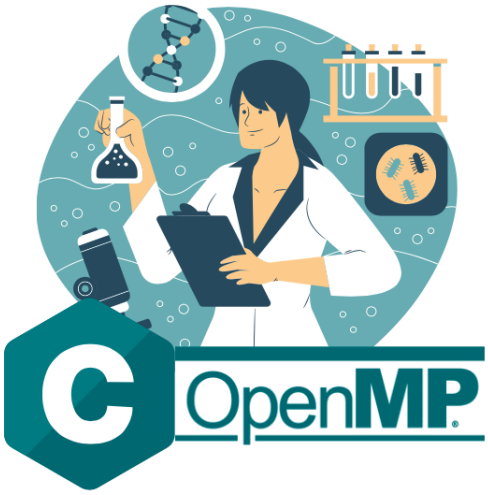
Arquitectura de Computadores
Facultad de Informática
UPV/EHU

Memoria

Genética



Iyán Álvarez
Iker Fernández

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

27 de diciembre de 2020

Índice general

1. Introducción

Este trabajo ha sido desarrollado en el tercer tema de la asignatura **Arquitectura de Computadores**, ofertada el 2º curso de Ingeniería Informática en la Facultad de Informática de la Universidad del País Vasco (UPV/EHU). Siendo este realizado en el curso 2020 - 2021 por Iyán Álvarez e Iker Fernández.

En el tercer tema de esta asignatura hemos analizado las características principales de los sistemas multiprocesador, utilizando OpenMP como herramienta para programar aplicaciones paralelas en estos sistemas. En clase hemos visto la teoría a la par que realizábamos varios ejercicios prácticos en los laboratorios.

Como proyecto final se nos ha encargado este trabajo, el cual tiene como objetivo aplicar todos los conceptos vistos en clase y trabajados en los laboratorios para desarrollar una aplicación paralela lo más eficiente posible. La aplicación inicialmente se tendrá que programar en serie, calcular sus tiempos, y aplicando las técnicas de paralelización reducir estos tiempos de ejecución.

En los laboratorios hemos utilizado un servidor DELL en el que cada estudiante tenemos nuestra cuenta de trabajo y el cual es accesible desde cualquier dispositivo utilizando la interfaz VPN de la universidad. Para conectarse con el multiprocesador se podía hacer desde una terminal Unix o a través de una aplicación como *PuTTy*. Sin embargo en nuestro grupo hemos visto más cómodo utilizar la aplicación CLion, puesto que tiene una interfaz más visual que una simple terminal. Finalmente todo el trabajo realizado, incluyendo las diferentes versiones de cada integrante del proyecto lo hemos subido a GitHub.

2. Fundamentos Teóricos

A la hora de hacer un proyecto de este nivel, es importante tener los conceptos teóricos bien interiorizados. Para ello hemos realizado a continuación un pequeño resumen de lo visto en clase para entender mejor el trabajo realizado.

2.1 Introducción

Uno de los objetivos de la arquitectura de computadores es acelerar la ejecución de los procesadores lo máximo posible. Hoy en día gracias al avance de la tecnología electrónica se pueden integrar millones de transistores en un chip. Eso se traduce en que los procesadores que existen a día de hoy son multicore, es decir, trabajan con varios núcleos.
Un computador paralelo es una máquina que está formada por muchas unidades de proceso que trabajan conjuntamente para resolver problemas grandes o bien, para trabajar para resolver problemas más rápido.
En esta asignatura solo hemos visto los sistemas SMP *(symmetric multiprocessor)* los cuales tienen un espacio de direccionamiento único y donde la comunicación entre procesos se realiza mediante variables compartidas.

2.2 Problemas principales de sistemas SMP

- **Influencia del código en serie:** La mayoría de veces no se puede ejecutar un programa entero en paralelo, una parte del programa se deberá ejecutar en serie.
- **Sincronización de variables compartidas:** Hay que sincronizar el uso de variables compartidas entre procesadores. Para ello hay dos opciones: secciones criticas (trozos de código que los procesos ejecutan de forma secuencial), y barreas (para sincronizar todos los procesos).
- **Reparto de trabajo:** Ha de ser equilibrado, para no perder tiempo. El reparto puede decidirse en tiempo de compilación (en caso de conocer el coste de tareas), o en tiempo de ejecución (en caso de que el coste de las tareas sea desconocido).

2.3 OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación de aplicaciones paralelas en máquinas de memoria compartida. Esta interfaz nos ofrece la identificación de los procesos paralelos, la declaración de variables (compartidas, privadas...), la definición de regiones paralelas y diferentes mecanismos de sincronización de procesos.
El modelo de programación paralela que se utiliza es Fork-Join. Los programas se ejecutan en un hilo principal y en un determinado momento, este hilo genera x hilos que se ejecutan en paralelo. Al finalizar la región paralela desaparecen todos los hilos menos el principal, el cual continúa su ejecución en serie.

Figura 2.1: Logo de OpenMP

2.3.1 Regiones paralelas

Es la parte del código que se ejecuta en paralelo.Esta parte del código será un bloque básico del programa. Se indica mediante la siguiente directiva:

#pragma omp parallel [cláusulas]

Para indicar el número de hilos se podrá indicar de diferentes maneras: de forma estática (antes de ejecutar el programa o utilizando una función de librería antes de la región paralela) o en tiempo de ejecución.
Entre los hilos las variables pueden ser privadas (private ó firstprivate) o compartidas (shared ó reduction). Este comportamiento se indica mediante sus correspondiente cláusula.

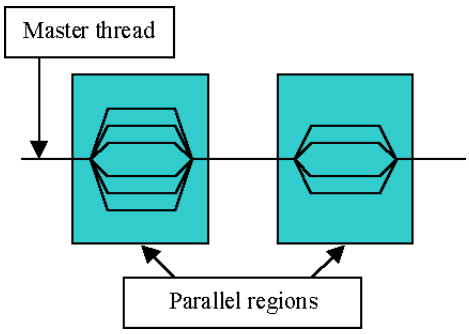


Figura 2.2: Ejemplo de las regiones paralelas

2.3.2 Reparto de trabajo

Hay que repartir las tareas entre los hilos de forma que el reparto sea lo más equilibrado posible. El reparto puede ser "manual" o a través de las opciones que ofrece OpenMP.
Se puede hacer un reparto en las iteraciones del bucle mediante la directiva pragma for y la cláusula schedule. El reparto puede ser estático o dinámico. También se pueden hacer secciones mediante la directiva pragma sections, pero en este trabajo no nos serán tan útiles.

2.3.3 Sincronización de los hilos

Cuando las tareas no son independientes y hay que utilizar variables compartidas, es necesario sincronizar la ejecución de los hilos. Para ello hay dos mecanismos de sincronización:

- **Secciones críticas** Será un trozo de código que no puede ser ejecutado por más de un hilo al mismo tiempo. Todos los hilos ejecutarán ese trozo de código, pero de uno en uno.

  #pragma omp critical

- **Barreras** Todos los procesos se quedan esperando hasta que todos lleguen a este punto. Las directivas parallel, for... tienen una barrera implícita al final.

  #pragma omp barrier

Con todo esto dicho tendremos las bases necesarias para realizar el trabajo.

3. Enunciado del Proyecto

A la vista de la situación que se ha dado con el COVID, para enfrentarse mejor ante nuevas pandemias, la OMS quiere hacer un análisis genético de un conjunto de muestras de las que dispone en sus laboratorios. La OMS dispone de un banco de datos con alrededor de 200.000 muestras genéticas de elementos patógenos. Cada una de ellas está identificada con 40 características genéticas y la OMS sabe qué tipo de enfermedad podría llegar a producir dicha muestra. Para ello, teniendo en cuenta 20 familias de posibles enfermedades, ha catalogado los tipos de enfermedades que puede ocasionar cada una de las muestras de laboratorio.
Dentro de un proyecto de investigación, nos han encargado procesar ese banco de datos para encuadrar cada muestra en 100 grupos genéticos distintos. La muestra se encuadra en un grupo genético en función de la cercanía de sus características genéticas, integrando en un grupo aquellas muestras con las mayores similitudes.Para clasificar a las muestras en su grupo genético se utilizará el algoritmo de clustering K-means.
El banco de datos contiene 40 datos genéticos de más de 200.000 muestras. El objetivo es agrupar todas esas muestras en 100 clústeres diferentes, en función de la cercanía entre sus 40 características genéticas.
Tras la generación de los 100 clústeres, se obtendrán unos resultados. Por una parte, se calcula un valor de densidad que viene a indicar el grado de dispersión de los elementos de un clúster. Por otra parte, se realiza un análisis de la presencia de las 20 enfermedades en las muestras encuadradas en cada grupo, obteniendo cuál es el mayor y el menor porcentaje de presencia de cada enfermedad entre todos los grupos y los grupos que tienen esos valores.
Los ficheros para desarrollar el proyecto son los siguientes:

- **gengrupos_s.c:** Programa principal de la aplicación.
- **fun_s.c:** Funciones a programar que se utilizaran en gengrupos_s.c.
- **defineg.h:** Definiciones para los ficheros gengrupos_s.c fun_s.c.
- **fun.h** Cabeceras de las funciones de fun_s.c.
- **dbgen.dat:** Fichero de entrada que contiene los datos genéricos de todas las muestras.
- **dbenf.dat:** Fichero de entrada que contiene 20 valores entre 0 y 1 por cada muestra.
- **res.out:** Fichero que contiene los resultados que debes conseguir.

Finalmente las funciones a programar y más adelantes a paralelizar serían las que contiene fun_s.c. A continuación una breve explicación de lo que hace cada una:

- **gendist:** Función para calcular la distancia genética entre dos elementos. Recibe dos vectores que contienen los 20 valores de sus características genéticas y se calcula la distancia euclídea entre ellos. Finalmente devuelve la distancia calculada.

- **grupo_cercano:** Función para asignar cada elemento del fichero su grupo genético más cercano. Recibe el número de elementos y dos matrices, una con los elementos a agrupar y otra con los centroides. Esta función hará uso de gendist para calcular las distancias.

- **calcular_densidad:** Función para calcular la densidad del grupo, es decir, la distancia media entre todos los elementos de cada centroide. Recibe una matriz con los elementos y una estructura de datos con un vector con la información de los grupos generados. Finalmente almacena la densidad en un vector.

- **analizar_enfermedades:** Función para realizar el análisis de enfermedades en los grupos, presencia máxima y mínima, y en qué grupos se dan esos valores máximos y mínimos. Recibe una estructura de datos con un vector con la información de los grupos generados y una matriz con todas las enfermedades. Finalmente almacena en una estructura de datos la información del análisis realizado.

4. Versión serie

En esta versión de la aplicación vamos a programar las funciones del fichero fun_s.c como hemos mencionado antes. Empezaremos desde 0 y nos vamos a enfocar en que los resultados sean correctos. En esta versión no nos importan los tiempos, tan solo los apuntaremos para luego compararlos con la versión paralela. Es importante entender y realizar correctamente lo que tenemos que hacer para luego a la hora de paralelizar ser lo más eficientes posible.

4.1 Solución

A continuación, el código final del fichero fun_s.c con las funciones implementadas.

4.1.1 gendist

```
1 double gendist (float *elem1, float *elem2) {
2     double acum = 0;
3     int i;
4     for (i = 0; i < NCAR; i++) {
5         double res = elem1[i] - elem2[i];
6         acum += pow(res, 2);
7     }
8     return sqrt(acum);
9 }
```

4.1.2 grupo_cercano

```
void grupo_cercano (int nelem, float elem[][NCAR], float cent[][NCAR], int *
    popul) {
    int ngrupo;
    double adis, dmin;
    for (int i = 0; i < nelem; i++) {
        dmin = DBL_MAX;
        for (int j = 0; j < NGRUPOS; j++) {
            adis = gendist(elem[i], cent[j]);
            if (adis < dmin) {
                dmin = adis;
                ngrupo = j;
            }
        }
        popul[i] = ngrupo;
    }
}
```

4.1.3 calcular_densidad

```
void calcular_densidad (float elem[][NCAR], struct lista_grupos *listag, float *
    densidad) {
    for (int i = 0; i < NGRUPOS; i++) {
        int nelem = listag[i].nelemg;
        if (nelem < 2) {
            densidad[i] = 0;
        }
        else {
            int actg;
            double acum = 0.0, cont = 0.0;
            for (int j = 0; j < nelem; j++) {
                actg = listag[i].elemg[j];
                for (int k = j+1; k < nelem; k++) {
                    int othg = listag[i].elemg[k];
                    acum += gendist(elem[actg], elem[othg]);
                    cont += 1.0;
                }
            }
            densidad[i] = (float) (acum/cont);
        }
    }
}
```

4.1.4 analizar_enfermedades

```
void analizar_enfermedades (struct lista_grupos *listag, float enf[][TENF],
    struct analisis *prob_enf) {
    for (int i = 0; i < TENF; i++) {
        float mediamin = FLT_MAX, mediamax = FLT_MIN;
        int gmax, gmin;
        for (int j = 0; j < NGRUPOS; j++) {
            int nelem = listag[j].nelemg;
            float acum = 0;
            for (int k = 0; k < nelem; k++) {
                int actg = listag[j].elemg[k];
                acum += enf[actg][i];
            }
            float mediaact = acum/nelem;
            if (mediaact < mediamin) {
                mediamin = mediaact;
                gmin = j;
            }
            else if (mediaact >= mediamax) {
                mediamax = mediaact;
                gmax = j;
            }
        }
        prob_enf[i].max = mediamax;
        prob_enf[i].min = mediamin;
        prob_enf[i].gmax = gmax;
        prob_enf[i].gmin = gmin;
    }
}
```

4.2 Decisiones

En esta versión no ha habido muchas complicaciones puesto que era un ejercicio de programación sencillo. La máxima complicación que hemos visto ha sido entender y analizar el objetivo de cada función, del programa principal y de las estructuras de datos a utilizar.

4.2.1 Control del tiempo

Hemos decidido añadir diferentes cláusulas de control de tiempo en el main para así determinar la cantidad de tiempo encesaria para cada tarea. De esta forma, podremos determinar de una manera más exacta las mejoras de la versión paralela.

4.3 Resultados

A medida que íbamos programando las funciones, hemos ido ejecutando distintas pruebas para corroborar el correcto funcionamiento de las mismas. Estas pruebas las hemos realizado mediante la ejecución del programa con 1000 elementos, de esta forma evitábamos largos tiempos de espera. Hemos decidido esto ya que disponíamos de un fichero con los resultados a obtener de 1000 elementos y del total.

4.3.1 Datos

Mediante la realización de distintas pruebas en la máquina, hemos obtenido los siguientes datos:

	Lectura	**Clustering**	**Ordenación**	**Densidad**	**Enfermedades**	**Escritura**	**TOTAL**
Media	3.193 s	168.624 s	0.020 s	28.809 s	0.003 s	0.000 s	200.674 s
Mínimo	3.058 s	160.965 s	0.019 s	26.106 s	0.002 s	0.000 s	190,15 s
Máximo	3.307 s	174.327 s	0.020 s	32.301 s	0.004 s	0.001 s	209,962 s

Es importante realizar varias pruebas puesto que no siempre se da el mismo resultado. En nuestro caso entre el mínimo y el máximo hay una diferencia de casi 19 segundos. Para la compilación de estos datos hemos realizado las pruebas en diferentes días y horas puesto que el servidor varía sus tiempos según la carga de trabajo que tenga.

4.3.2 Gráficas

Para ver de una manera más visual los tiempos de ejecución de la aplicación hemos realizado una gráfica en forma circular. Como se puede apreciar, el tiempo de clustering ocupa más del 80 % del tiempo total de la versión serie. Por tanto optimizando el código que se ocupa de ese tarea, entre ello la función grupo_cercano; obtendrán un factor de aceleración elevado tras una correcta paralelización.

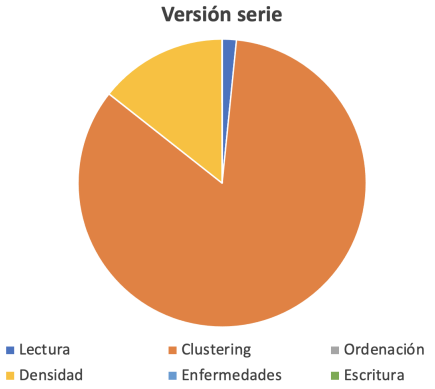


Figura 4.1: Tiempo de ejecución de las diferentes tareas en la versión serie

5. Versión paralela

En esta versión de la aplicación vamos a paralelizar las funciones anteriormente programadas en el fichero fun_s.c. Para ello tendremos que analizar qué bucles y secciones son las más costosas y analizar cual es la mejor manera de optimizarlas al máximo. Nuestro objetivo en esta parte del trabajo es reducir considerablemente los tiempos de ejecución. También compararemos los tiempos cambiando el número de hilos, el reparto... y más tarde valoraremos cual es la mejor combinación.

5.1 Solución

5.1.1 grupo_cercano

A la hora de ejecutar la versión serie, esta era la función más costosa de todas y la que mejores resultados nos iba a dar al paralelizarla. La paralelización se ha realizado en el primer bucle, por tanto esta será por cada elemento de la matriz.
Después de realizar varias pruebas con los repartos y los hilos. Obtuvimos los mejor resultados con dynamic,2 y 32 hilos.

```
1  void grupo_cercano (int nelem, float elem[][NCAR], float cent[][NCAR], int *
       popul) {
2       ...
3       #pragma omp parallel for private(i, j, adis, dmin, ngrupo) schedule(dynamic
       ,2) num_threads(32)
4       for (i = 0; i < nelem; i++) {
5           dmin = DBL_MAX;
6           for (j = 0; j < NGRUPOS; j++) {
7               adis = gendist(elem[i], cent[j]);
8               if (adis < dmin) {
9                   dmin = adis;
10                  ngrupo = j;
11              }
12          }
13          popul[i] = ngrupo;
14      }
15 }
```

5.1.2 calcular_densidad

La segunda función más costosa de la versión serie es la que calcula la densidad del grupo. En este caso hicimos uso de dos variables de reducción, acum y cont, para luego usarlas en el calculo de densidad.
Al igual que en grupo_cercano, realizamos varias pruebas con los repartos y los hilos; obteniendo los mejor resultados con dynamic,2 y 32 hilos.

```
void calcular_densidad (float elem[][NCAR], struct lista_grupos *listag, float *
    densidad) {
    ...
    #pragma omp paralell for private(j, k, actg, othg) reduction(+:acum, cont)
    schedule(dynamic, 2) num_threads(32)
    for (j = 0; j < nelem; j++) {
        actg = listag[i].elemg[j];
        for (k = j+1; k < nelem; k++) {
            othg = listag[i].elemg[k];
            acum += gendist(elem[actg], elem[othg]);
            cont += 1.0;
        }
    }
    ...
}
```

5.1.3 analizar_enfermedades

El análisis de enfermedades en la versión serie necesitaba un tiempo muy reducido, por tanto sabíamos que iba a ser difícil mejorar ese tiempo notablemente. Aún así, como el objetivo de este trabajo era aprender y practicar los mecanismos de paralelización, decidimos realizarla en el bucle que recorría los elementos de los grupos. Hacemos uso de una variable de reducción, acum, para el posterior cálculo de la media.
Hemos determinado el reparto static con 2 hilos, de esta forma evitamos generar un número elevado de hilos cargando la máquina innecesariamente.

```
void analizar_enfermedades (struct lista_grupos *listag, float enf[][TENF],
    struct analisis *prob_enf) {
    ...
    #pragma omp parallel for private(k, actg, mediaact) shared(mediamin,
    mediamax, gmin, gmax) reduction(+ : acum) schedule(static,1) num_threads(32)
    for (k = 0; k < nelem; k++) {
        actg = listag[j].elemg[k];
        acum += enf[actg][i];
    }
    ...
}
```

5.1.4 main

En la parte del programa principal también había varios bucles y secciones que se podían paralelizar. Al igual que con analizar_enfermedades, no iba a haber una gran reducción de tiempo, sin embrago como hemos explicado antes el objetivo era aprender y practicar.

Primero de todo, nos encontramos con el proceso de lectura de los ficheros, este proceso no se puede paralelizar ya que se van leyendo los datos una a uno y guardando en la matriz. Además no es posible indexar los accesos a un fichero, como sí podríamos hacer con un array.

Posteriormente, nos encontramos con el proceso de clustering, en el que se generan los primeros centroides y se reorganizan en grupos.
La primera parte que paralelizamos fue la que clasificaba los elementos y calculaba los nuevos centroides. Para ello abrimos una sección paralela con 32 hilos, y paralelizamos el bucle que anulaba todos los valores de la matriz additions, y seguidamente introducíamos una sección single para el bucle que acumula los valores de cada característica. Finalmente hemos paralelizado el bucle que calcula los nuevos centroides y que decide si el proceso ha terminado o no.

Una vez acabado, viene el proceso de ordenación, uno de los for's lo hemos podido paralelizar correctamente, en cambio hemos determinado que el siguiente no se puede paralelizar debido a la dependencia sucesiva de los factores a utilizar en el bucle, es decir si lo paralelizáramos, todas las instrucciones irían en una sección crítica.

Después nos encontramos con el proceso de cálculo de densidad y análisis de enfermedades, que son llamadas a las funciones previamente explicadas.

Por último, nos encontramos con el proceso de escritura, que al igual que la lectura no se puede paralelizar por el accesoa a ficheros.

```
num_ite = 0; fin = 0;
while ((fin == 0) && (num_ite < MAXIT)) {
    grupo_cercano(nelem, elem, cent, popul);
    #pragma omp parallel num_threads(32)
    {
        #pragma omp  for private(i, j)  schedule(static)
        for (i = 0; i < NGRUPOS; i++) {
            for (j = 0; j < NCAR + 1; j++) {
                additions[i][j] = 0.0;
            }
        }
        #pragma omp single
        {
            for (i = 0; i < nelem; i++) {
                for (j = 0; j < NCAR; j++) {
                    additions[popul[i]][j] += elem[i][j];
                }
                additions[popul[i]][NCAR]++;
            }
            fin = 1;
        }

        #pragma omp for private(i, j, discent) schedule(static)
        for (i = 0; i < NGRUPOS; i++) {
            if (additions[i][NCAR] > 0) {
                for (j = 0; j < NCAR; j++) {
                    newcent[i][j] = additions[i][j] / additions[i][NCAR];
                }
                discent = gendist (&newcent[i][0], &cent[i][0]);
                if (discent > DELTA) {
                    fin = 0;
                }
                for (j = 0; j < NCAR; j++) {
                    cent[i][j] = newcent[i][j];
                }
            }
        }
        #pragma omp single
        {
            num_ite++;
        }
    }
...
}
#pragma omp parallel for private(i) schedule(static)
for (i = 0; i < NGRUPOS; i++) {
    listag[i].nelemg = 0;
}
```

5.2 Decisiones

A la hora de realizar el proyecto hemos tenido que tomar algunas decisiones y mediantes distintas pruebas determinar cuál era la más adecuada. A parte de esto, hemos podido determinar algunos problemas.

5.2.1 Decisión sobre schedule e hilos

Para decidir la mejor opción para paralelizar nuestro programa hemos recogido una gran cantidad de datos, con distintos schedule y número de hilos.
Una vez recogidos todos los datos, los analizamos y determinamos la mejor opción para cada tarea.
Posteriormente, determinamos en el programa las opciones seleccionadas y realizamos diversas pruebas para comprobar que el speed-up obtenido era el mejor posible. y Además hemos utilizado el menor número de hilos si el tiempo obtenido era el mismo, de esta forma hemos obtenido una solución rápida y eficiente.

5.2.2 Problemas con máximo de hilos

Hemos realizado pruebas con 2, 4, 8, 16, 24, 32, 62 y 64 hilos (62 y 64 no incluidos en el informe de tablas). A pesar de que se nos informó de que la máquina podía no desenvolverse correctamente con 64 hilos, hemos querido realizar nuestras pruebas concluyendo que no solamente falla con 64, sino que con un número aproximado al límite el tiempo de la mayoría de ejecuciones aumenta drásticamente.

5.2.3 Problemas con la máquina remota

A la hora de ejecutar demasiadas pruebas al mismo tiempo, la máquina se saturaba, causando tiempos altos e incoherentes. Desde la página *dif-cluster.si.ehu.es* hemos podido visualizar el estado del servidor y conocer la carga de la CPU para obtener los resultados más fieles posibles a la realidad. Por esto, hemos realizado las pruebas en horas en las que no había un gran número de personas ejecutando sus programas.

En la siguiente imagen se puede visualizar la carga de la CPU, esto ralentizaba las ejecuciones.

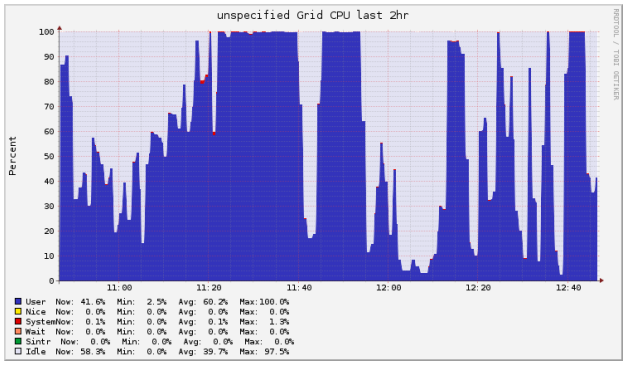


Figura 5.1: Saturación de la CPU

5.3 Resultados

Al igual que con la versión serie, hemos hecho varias pruebas de rendimiento para ver cual era la configuración más eficiente. Puesto que hay diferentes configuraciones para la paralelización (número de hilos, reparto de trabajo...) hemos hecho diferentes pruebas con todas las variantes para más tarde hacer un análisis de cuales eran las más eficientes y crear la versión paralela más rápida. En esta versión todas las pruebas las hemos ido haciendo con 25.000 elementos, de esta forma las ejecuciones eran más rápidas.

5.3.1 Datos

Aquí se pueden encontrar los resultados de todas las pruebas realizadas sobre el número de elementos total, en los siguientes 6 schedule's. Cada uno de estos tiene las distintas ejecuciones realizadas con 2, 4, 8, 16, 24 y 32 hilos. Además se ha realizado la prueba con 64 hilos, debido a los problemas expuestos previamente no las hemos incluido en las tablas.

Como conclusión podríamos ver que el mayor factor de aceleración lo tiene la configuración de reparto de tipo dynamic,2 con 32 hilos. Tras haber realizado todas estas pruebas con la cláusula *schedule(runtime)* decidimos especificar las mejores opciones directamente en el código, sin necesidad de especificar ninguna variable de entorno.

Leyenda

- **Casillas amarillas:** el proceso paralelizado.
- **Casillas verde claro:** mejor tiempo de paralelización del proceso.
- **Casillas verde oscuro:** mejor speed-up de todos los procesos.

5.3.1.1 static

Total (static)							
	Serie	2	4	8	16	24	32
Lectura	3,196	3,307	3,305	3,316	3,305	3,313	3,371
Clustering	197,003	85,542	43,596	25,279	12,039	8,594	8,102
Ordenación	0,019	0,010	0,006	0,004	0,004	0,004	0,005
Densidad	28,638	21,832	12,791	6,954	3,626	2,418	1,832
Enfermedades	0,003	0,003	0,003	0,003	0,003	0,003	0,003
Escritura	0,000	0,000	0,000	0,000	0,000	0,000	0,000
TOTAL	228,859	110,694	59,701	35,556	18,977	14,332	13,313
SPEED-UP	-	2,067	3,833	6,437	12,060	15,968	17,191

Figura 5.2: Tiempos de ejecución con el schedule static

5.3.1.2 static,1

Total (static,1)							
	Serie	2	4	8	16	24	32
Lectura	3,196	3,308	3,314	3,316	3,313	3,432	3,306
Clustering	197,003	89,611	43,856	22,637	12,071	9,664	7,062
Ordenación	0,019	0,014	0,008	0,005	0,005	0,007	0,009
Densidad	28,638	14,328	7,162	3,644	1,846	1,246	0,931
Enfermedades	0,003	0,003	0,003	0,003	0,003	0,003	0,003
Escritura	0,000	0,000	0,000	0,000	0,000	0,000	0,000
TOTAL	228,859	107,264	54,343	29,605	17,238	14,352	11,311
SPEED-UP	-	2,134	4,211	7,730	13,276	15,946	20,233

Figura 5.3: Tiempos de ejecución con el schedule static,1

5.3.1.3 static,2

Total (static,2)							
	Serie	2	4	8	16	24	32
Lectura	3,196	3,312	3,315	3,312	3,305	3,307	3,311
Clustering	197,003	86,007	43,749	22,511	11,962	8,707	6,919
Ordenación	0,019	0,013	0,007	0,004	0,004	0,005	0,005
Densidad	28,638	14,338	8,612	3,598	1,827	1,257	0,958
Enfermedades	0,003	0,003	0,003	0,003	0,003	0,003	0,003
Escritura	0,000	0,000	0,000	0,000	0,000	0,000	0,000
TOTAL	228,859	103,673	55,686	29,428	17,101	13,279	11,196
SPEED-UP	-	2,208	4,110	7,777	13,383	17,235	20,441

Figura 5.4: Tiempos de ejecución con el schedule static,2

5.3.1.4 dynamic

Total (dynamic)							
	Serie	2	4	8	16	24	32
Lectura	3,196	3,312	3,312	3,311	3,310	3,312	3,332
Clustering	197,003	88,621	46,393	23,149	12,170	8,573	7,119
Ordenación	0,019	0,014	0,008	0,005	0,005	0,007	0,008
Densidad	28,638	15,202	7,428	2,565	1,798	1,207	0,940
Enfermedades	0,003	0,003	0,003	0,003	0,003	0,003	0,003
Escritura	0,000	0,000	0,000	0,000	0,000	0,000	0,000
TOTAL	**228,859**	**107,152**	**57,144**	**29,033**	**17,286**	**13,102**	**11,402**
SPEED-UP	-	**2,136**	**4,005**	**7,883**	**13,240**	**17,467**	**20,072**

Figura 5.5: Tiempos de ejecución con el schedule dynamic

5.3.1.5 dynamic,1

Total (dynamic,1)							
	Serie	2	4	8	16	24	32
Lectura	3,196	3,312	3,315	3,443	3,305	3,307	3,311
Clustering	197,003	93,194	45,990	23,208	12,199	8,556	6,877
Ordenación	0,019	0,014	0,008	0,005	0,005	0,007	0,009
Densidad	28,638	14,309	7,136	3,583	1,801	1,207	0,913
Enfermedades	0,003	0,003	0,003	0,003	0,003	0,003	0,003
Escritura	0,000	0,000	0,000	0,000	0,000	0,000	0,000
TOTAL	**228,859**	**110,832**	**56,452**	**30,242**	**17,313**	**13,08**	**11,113**
SPEED-UP	-	**2,065**	**4,054**	**7,568**	**13,219**	**17,497**	**20,594**

Figura 5.6: Tiempos de ejecución con el schedule dynamic,1

5.3.1.6 dynamic,2

Total (dynamic,2)							
	Serie	2	4	8	16	24	32
Lectura	3,196	3,306	3,306	3,306	3,306	3,313	3,306
Clustering	197,003	87,206	44,205	22,728	12,064	8,408	6,667
Ordenación	0,019	0,013	0,007	0,005	0,004	0,005	0,005
Densidad	28,638	14,278	7,143	3,583	1,794	1,202	0,909
Enfermedades	0,003	0,003	0,003	0,003	0,003	0,003	0,003
Escritura	0,000	0,000	0,000	0,000	0,000	0,000	0,000
TOTAL	**228,859**	**104,806**	**54,664**	**29,625**	**17,171**	**12,931**	**10,89**
SPEED-UP	-	**2,184**	**4,187**	**7,725**	**13,328**	**17,698**	**21,016**

Figura 5.7: Tiempos de ejecución con el schedule dynamic,2

5.3.2 Gráficas

Para ver de una manera más visual los tiempos de ejecución de la aplicación hemos realizado una gráfica en forma circular. En este caso, como la lectura no se puede paralelizar, su porcentaje de tiempo ha subido drásticamente, ocupando algo más del 25 % del tiempo total de la ejecución del programa. La parte del clustering sigue siendo la parte que más tiempo lleva, sin embargo esta vez ocupa al rededor del 60 %, aproximadamente un 20 % menos que en la versión serie.
Como conclusión podríamos decir que hemos conseguido reducir el tiempo de densidad y clustering considerablemente. En el caso de la densidad, hay un factor de aceleración del 31,69 % respecto a la versión serie. Por otro lado la parte del clustering ha obtenido un 25,29 % de aceleración.

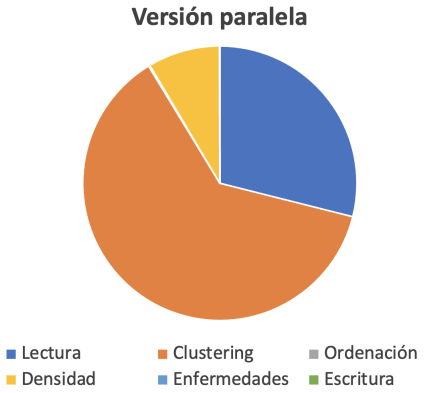


Figura 5.8: Tiempo de ejecución de las diferentes tareas en la versión paralela

6. Comparativa

Una vez desarrolladas ambas versiones, es importante compararlas para apreciar la reducción considerable en el tiempo de ejecución. A continuación se puede observar una breve explicación razonada sobre los procesos y los cambios realizados de forma justificada. De esta forma es más fácil comprender la gráfica comparativa en la siguiente página.

Lectura
El proceso de lectura necesita aproximadamente el mismo tiempo en ambas versiones. Esto se debe a que este proceso no ha cambiado; ya que no es posible paralelizarlo.

Clustering
El proceso de clustering, es la tarea que más tiempo necesita en la versión serie, y una vez paralelizada, es la que obtiene los mejores resultados de todo el programa. En la gráfica se puede apreciar de una forma más visual la bajada drástica del tiempo de ejecución de esta tarea y del total.

Ordenación
El proceso de ordenación es un proceso que apenas necesita tiempo para ejecutarse, pero paralelizándolo hemos obtenido pequeñas mejoras (milésimas de segundo). En cuánto al cómputo total del programa esto no aporta un gran mejora al tiempo total de ejecución.

Densidad
El proceso de cálculo de la densidad de los centroides es el segundo proceso que más tiempo lleva. Al igual que nos pasa con el proceso de clustering, su tiempo de ejecución mejora mucho al paralelizarlo. Esto nos proporciona bajada considerable en el tiempo de ejecución de la tarea y del total.

Enfermedades
El proceso de análisis de enfermedades necesita aproximadamente el mismo tiempo en ambas versiones. Esto se debe a que este proceso apenas necesita tiempo para ejecutarse, por lo que las mejoras son minúsculas. A su misma vez ocupa un tiempo de ejecución muy reducido en el cómputo total del programa.

Escritura
Al igual que el proceso de lectura, la escritura necesita aproximadamente el mismo tiempo en ambas versiones; ya que no es posible paralelizarlo.

En la siguiente gráfica se puede observar la diferencia entre ambas versiones, en el eje X de la gráfica se pueden apreciar las distintas tareas del programa, siendo la última barra la suma de los procesos anteriores. En el eje Y se puede apreciar la cantidad de segundos necesaria para la ejecución de la tarea. En azul se puede observar el tiempo necesario para la ejecución de la versión serie; y en naranja, para la versión paralela.

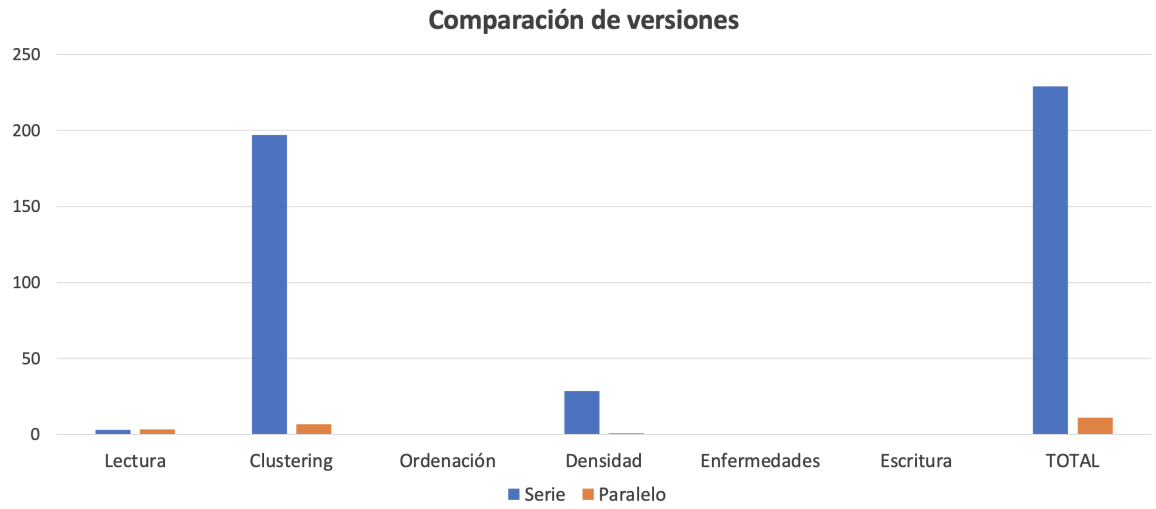


Figura 6.1: Tiempo de ejecución de las diferentes versiones

7. Conclusiones

En conclusión, este trabajo realizado a lo largo del último mes nos sirve para aprender, practicar e interiorizar los conceptos aprendidos del tema *Paralelización* de la asignatura Arquitectura de Computadores. Tras la realización de este proyecto hemos visto la importancia de la paralelización en problemas de la vida real y las mejoras que esta nos aporta. No solo hemos visto la importancia de hacer un código limpio y eficiente, sino que hemos aprendido que mediante mecanismos y API's como OpenMP, podemos hacer que nuestros futuros proyectos sean mucho más rápidos y eficientes.

Antes de realizar este trabajo ambos integrantes del grupo no sabíamos como de eficaz iba a ser paralelizar nuestro programa. Tras varias pruebas hemos obtenido un tiempo de ejecución 20 veces mejor comparándolo con la versión serie.

A lo largo de este trabajo hemos tenido varios errores y problemas. Los hemos logrado superar con éxito. Son esta clase de obstáculos los que consideramos que nos hacen aprender y que a futuro tendremos en cuenta para no volver a realizar. Sí tuviésemos que realizar este trabajo de nuevo, lo más probable es que nos llevase mucho menos tiempo del que hemos necesitado en esta ocasión.

Ambos integrantes del grupo consideramos que hemos realizado un buen trabajo y estamos muy contentos con este. Creemos que hemos logrado con creces el objetivo del mismo y lo hemos conseguido documentar de una manera clara y fácil de entender.

8. Bibliografía

- **eGela (UPV/EHU) - Arquitectura de Computadores**
  Transparencias Paralelismo
  Resumen OpenMP
  Ejercicios OpenMP

- **StackExchange**
  C code to add in the document
  listing: how to set code color and frame color

- **OpenMP Forum**

- **JetBrains**
  Remote server configuration
  Remote host tool window

- **HTMLcolorcodes**
  Códigos de colores HTML