PRÁCTICA 1

Propuesta y estudio de la paralelización

Autores:

Juan Lopez Quiles
Óscar Falcó Herrera
Manuel Garcia Cremades
Alejandro Aliaga Hyder

ÍNDICE

1 Presentación del problema propuesto
2 Revisar problemas y aplicaciones derivadas…
3 Estudiar la codificación del programa
3.1 Acceso y variables
3.2 Problemas con la caché
3.3 Variar la carga
4 Proponer arquitecturas idóneas para la paralelización
5 Defender la paralelización del programa
5.1 Grafo de control de flujo (GCF)
5.2 Trozos paralelizables revelados en el GCF
6 Métodos de estimar la ganancia
7 Análisis del rendimiento
8 Parámtros de compilación
8.1 Parámetros de compilación usados
9 Bibliografía

1. Presentación del problema propuesto

Hemos propuesto un problema de interpolación de imágenes, un problema en el cual vamos a aumentar el tamaño de una imagen. Nuestro programa será capaz de crear esta nueva imagen sin que se pixele, gracias a que irá creando los nuevos píxeles, interpolando entre los píxeles originales que tenga a su alrededor y consiguiendo un color que esté entre estos píxeles originales. Por así decirlo, un color que sea una transición entre los originales.

La fórmula que hemos usado es la siguiente:

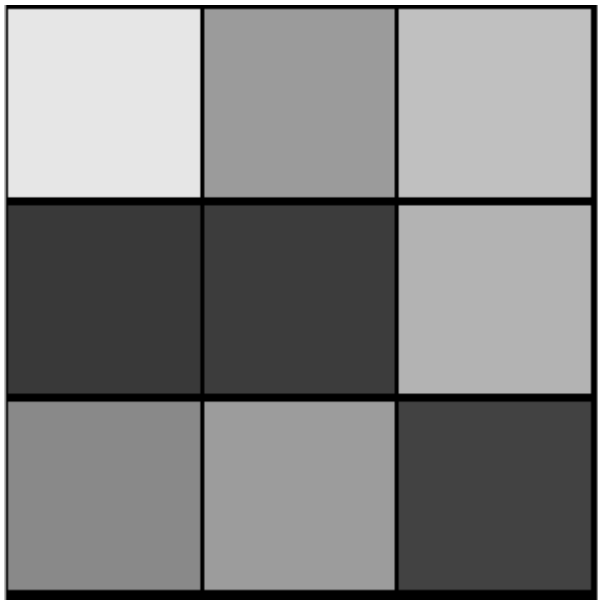
Interpolación bilineal de una variable independiente:

$$f(x,y) \approx f(0,0)\,(1-x)(1-y) + f(1,0)\,x(1-y) + f(0,1)\,(1-x)y + f(1,1)xy.$$

Usamos esta opción, dado que tenemos un sistema de coordenadas en el cual tenemos 4 puntos conocidos de f, que será nuestra imagen original.

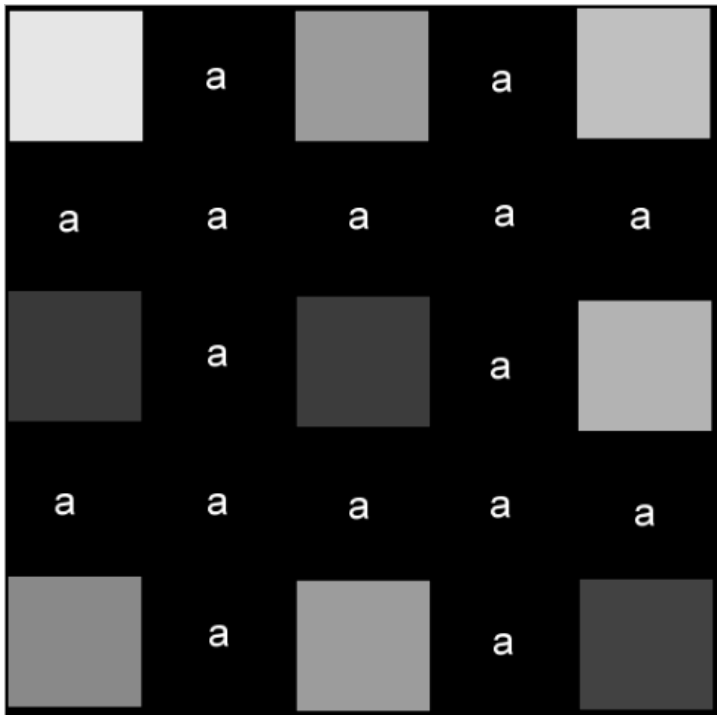
Un pequeño ejemplo de nuestro problema sería el siguiente:

Nosotros tendremos una imagen original, la cual buscamos ampliar.



A continuación, para cada nuevo píxel que en este caso hemos llamado "a", interpolaremos entre 2

píxeles de la antigua imagen. Existirán píxeles "a" los cuáles dependerán de los píxeles diagonales, dado que SOLO dependemos de los píxeles originales en nuestro problema. Los nuevos píxeles, los píxeles "a" tendrán un color intermedio entre los 2 colores de los píxeles originales.



Mediante la fórmula anteriormente explicada, obtendremos un valor matemático que será el color final. Dada la explicación del proceso, un pequeño ejemplo seria el siguiente:

Dada la explicación del proceso, un pequeño ejemplo seria el siguiente:

2. Revisar problemas y aplicaciones derivadas...

- **Revisar problemas y aplicaciones derivadas de cualquier rama del conocimiento, actuales o no, cuya carga computacional sea tan elevada como para justificar acometer un proceso de paralelización. Elegir una aplicación secuencial candidata a ser paralelizada.**

La práctica que se nos propone, tiene como objetivo la búsqueda y selección de aplicaciones secuenciales cuya carga computacional sea lo suficientemente alta como para sugerir su paralelización.

La utilización de todos los recursos disponibles y la capacidad de utilizar un mayor poder de computación para resolver problemas complejos o de grandes dimensiones, son las características más destacables a la hora de seleccionar aplicaciones para su paralelización.

En nuestro caso, será el re escalado de imágenes la aplicación que trataremos y será objetivo de paralelización con posterioridad. Existen distintos algoritmos que pueden ser utilizados para el re escalado de imágenes, siendo la interpolación bilineal el que trataremos en nuestra practica.

La interpolación bilineal consiste en la interpolación del valor de los colores de los cuatro pixeles adyacentes a los nuevos pixeles a generar en el aumento del tamaño una imagen, para poder conocer el color de estos. Según la posición y disposición de los pixeles conocidos, los nuevos pixeles pueden usar tanto los ejes diagonales como los ejes-xy. Dado su funcionamiento podemos observar que es una aplicación paralelizable al consistir básicamente en el cálculo del valor del color de todos los nuevos pixeles generados.

3. Estudiar la codificación del programa FALTA

3.1 Acceso y variables

Nuestra aplicación depende de una imagen que debe ser proporcionada por el usuario y que será la imagen a re escalar. Una vez proporcionada la imagen podemos se procederá al re escalado.

Las variables que usa la aplicación son usadas para guardar el valor de los pixeles de las imágenes. En concreto usa cuatro colores: rojo (red), blue (azul), green (verde) y alpha, que es lo que se conoce como la codificación de colores RGBA, que darán color a la imagen.

3.2 Problemas con la caché

Podemos prever problemas en la memoria cache si la variable introducida para la creación de la nueva imagen es muy grande, el tamaño de la imagen será muy grande (pudiendo hablar de varios gigabytes) y necesitando nuestro ordenador tener que hacer un gran uso de la memoria RAM.

3.3 Variar la carga

la aplicación permite variar el tamaño y con ello su peso, según el peso de las imágenes introducidas, lo que permite variar la carga. Cuanto mayor sea, mayor será el tamaño de la imagen y mayor será tanto el peso como la distorsión de la imagen creada por la aplicación, pudiendo llegar a valores de varios gigabytes.

4. Proponer arquitecturas idóneas para la paralelización

La clasificación más popular de computadores es la taxonomía de Flynn. Esta taxonomía de las arquitecturas está basada en la clasificación teniendo en cuenta el flujo de datos e instrucciones en un sistema.

La taxonomía de Flynn distingue **cuatro tipos de arquitecturas:**

SISD: Flujo único de instrucciones y flujo único de datos.

SIMD: Flujo único de instrucción y flujo múltiple de datos. Es este tipo el que con mas probabilidad sea el más idóneo para nuestro problema.

MISD: Múltiples instrucciones y flujo único de datos.

MIMD: Múltiples instrucciones y Múltiples flujos de datos.

También podemos hablar de la estructura lógica del programa (arquitectura de software), que determinara el grado de paralelización que puede llevarse a cabo en la aplicación. En ella podríamos destacar la dependencia de datos, ya que en nuestra aplicación el cálculo de los valores de los pixeles pueden venir determinados, tanto por los pixeles en las diagonales como por aquellos que aparecen en los ejes-xy según su disposición. Estos problemas podrían resolverse limitando los parámetros de entrada de la aplicación y en algunos casos a costa de empeorar el resultado final (algunos casos solo permiten cálculo de los pixeles en el eje-x).

5. Defender la paralelización del programa

Como se observa, nuestra aplicación es paralelizable tanto a nivel de datos, ya que para el calculo de cada pixel se tendrá que tener en cuenta los distintos pixeles adyacentes según la posición del nuevo pixel (además de su eje según los pixeles conocidos), como a nivel funcional ya que distintas tareas (cálculos iguales o distintos) pueden ser llevados simultáneamente (de forma paralela).

Algunas de las métricas que nos aportan información sobre su posible paralelización son:

El tiempo de ejecución: Métricas como función del tamaño del problema, procesadores disponibles, número de procesos, variables dependientes del algoritmo y las de características del hardware sobre el que se ejecuta.

Ley de Amdahl: que nos dice que existe una parte secuencial del programa que no podemos paralelizar (cota inferior del tiempo de ejecución) y una parte paralelizable mediante la cual reduciremos el tiempo de ejecución. Su complejidad crece con el numero de procesadores a usar.

Ley de Gustafson (problemas escalados, donde el tamaño del problema crece): que establece que cualquier problema suficientemente grande puede ser eficientemente paralelizado.

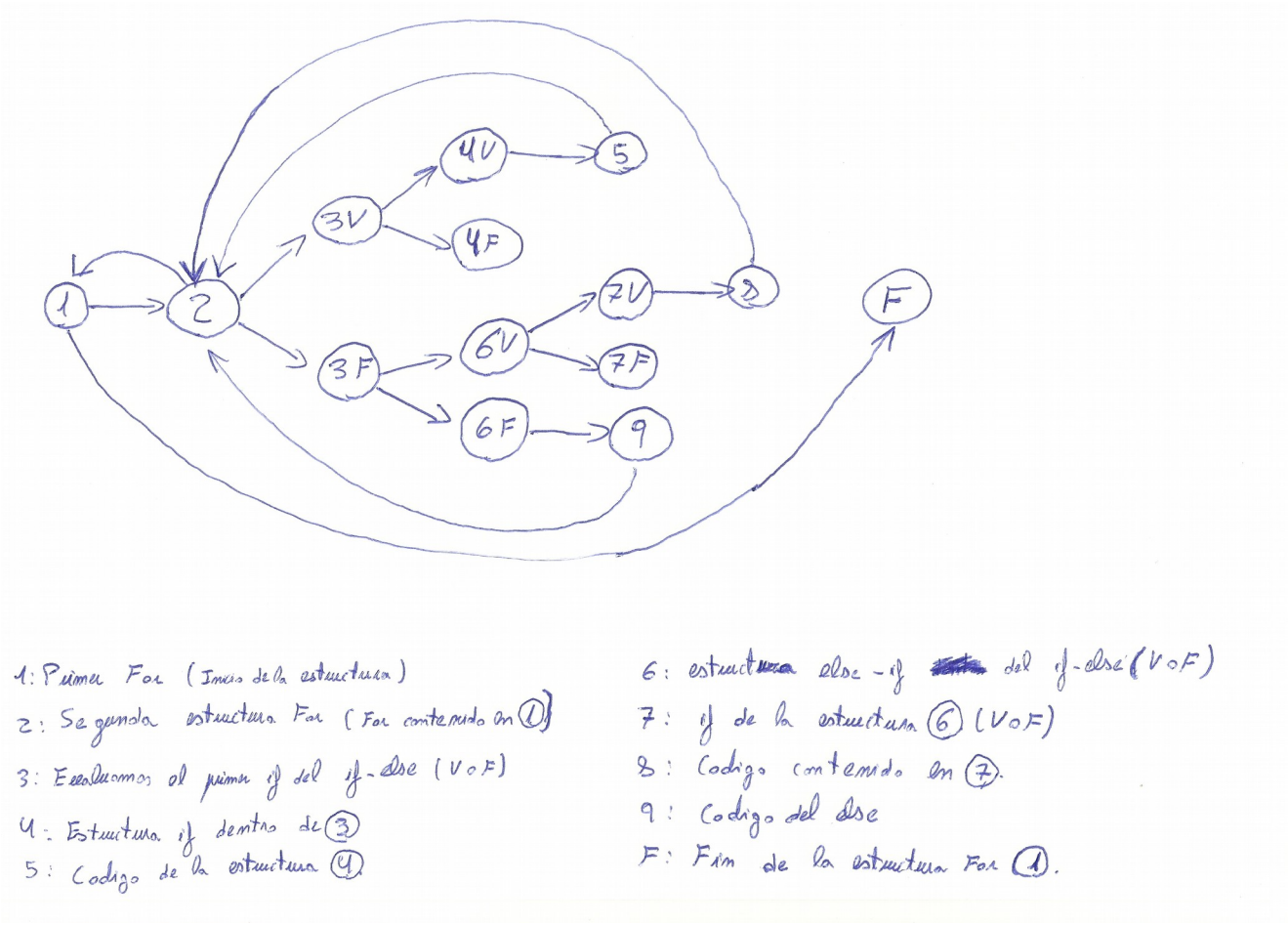
También podemos tener en cuenta la granularidad: Conforme aumenta la granularidad, disminuye el grado de paralelismo.

La contribución de la arquitectura al aumento de las prestaciones de los computadores ha venido de la mano del paralelismo y del aprovechamiento de la localidad de acceso a los datos.

Las principales razones para la construcción de máquinas paralelas son: disminuir el tiempo total de ejecución de una aplicación, conseguir resolver problemas más complejos, de grandes dimensiones, y proporcionar concurrencia, o sea, permitir la ejecución simultánea de tareas.

Podemos considerar como programación paralela la que se hace usando varios procesadores para acelerar la resolución de los problemas con que trabajemos. Trabajaremos con algoritmos paralelos, donde se especifica la forma en que los distintos elementos de proceso comparten los datos, y cómo colaboran en la resolución de los problemas. En general consideraremos que los procesadores son todos del mismo tipo, con lo que tendremos sistemas paralelos homogéneos.

5.1 Grafo de control de flujo (GCF)



5.2 Trozos paralelizables revelados en el GCF

Las partes de nuestro código que son paralelizables pueden verse en nuestro grafo de control de flujo, en concreto, podemos hablar de el código que contienen las estructuras representadas con 5 y 9 que contienen los cálculos que serán paralelizados con posterioridad. Como ya hemos explicado anteriormente nuestro programa debe calcular el valor del nuevo pixel con la interpolación de los valores de los 4 pixeles situados en las diagonales o en los ejes-xy, este cálculo se debe hacer para cada nuevo pixel y será aquel con más grado de paralelización.

5:

```
RGBApixel pixel1 = OriginalImage.GetPixel(i/2,j/2);
RGBApixel pixel2 = OriginalImage.GetPixel((i+1)/2,j/2);
RGBApixel pixel3 = OriginalImage.GetPixel(i/2,(j+1)/2);
RGBApixel pixel4 = OriginalImage.GetPixel((i+1)/2,(j+1)/2);

RGBApixel tmp = bilinealInterpolation( pixel1,pixel2,pixel3,pixel4,0.5,0.75 );
```

6:

```
RGBApixel pixel1 = OriginalImage.GetPixel(i/2,j/2);
RGBApixel pixel2 = OriginalImage.GetPixel((i+1)/2,j/2);
RGBApixel pixel3 = OriginalImage.GetPixel(i/2,(j+1)/2);
RGBApixel pixel4 = OriginalImage.GetPixel((i+1)/2,(j+1)/2);

RGBApixel tmp = bilinealInterpolation( pixel1,pixel2,pixel3,pixel4,0.25,0.5 );
```

```
int newRed = pixela1.Red * (1 - x) * ( 1 - y) +
        pixelb1.Red * (x) * (1 - y) +
        pixela2.Red * (1-x) * (y) +
        pixelb2.Red * x * y;

int newGreen = pixela1.Green * (1 - y) * ( 1 - y) +
        pixelb1.Green * (x) * (1 - y) +
        pixela2.Green * (1-x) * (y) +
        pixelb2.Green * x * y;

int newBlue = pixela1.Blue * (1 - y) * ( 1 - y) +
        pixelb1.Blue * (x) * (1 - y) +
        pixela2.Blue * (1-x) * (y) +
        pixelb2.Blue * x * y;


int newAlpha = pixela1.Alpha * (1 - y) * ( 1 - y) +
        pixelb1.Alpha * (x) * (1 - y) +
        pixela2.Alpha * (1-x) * (y) +
        pixelb2.Alpha * x * y;
```

6. Métodos de estimar la ganancia

Medidas de rendimiento:

Eficiencia:
Cuando hablamos de medición de la eficiencia para esta asignatura, debemos recordar lo visto en su predecesora Arquitectura de los computadores. En esta ultima vimos como podíamos medir la eficiencia mediante el tiempo de ejecución (o respuesta) de una aplicación y la productividad (número de aplicaciones por unidad de tiempo).

En el material proporcionado en esta nueva asignatura podemos encontrar las siguientes formulas para su cálculo, haciendo uso de la ganancia, además de un repaso a los citados métodos vistos en AC.

$$E_P = \frac{G_p}{P}; \qquad E_P \leq 1$$

También podemos introducir nuevas conceptos como el speed-up, el speed-up es usado para indicar el grado de ganancia de velocidad de una computación paralela, mientras la eficiencia mide la porción útil del trabajo total realizado por los procesadores.

El speed-up es usado en los cálculos de la ley de Amdahl y la ley de Gustafson, anteriormente citadas. La parte paralelizable de la aplicación debe ser lo mayor posible según estas leyes, dado que es en esta donde se produce la mejora del rendimiento.

Ganancia:
Es la medida que usamos para representar el aumento de velocidad obtenido en la parte paralelizable del problema (cuanto mayor mejor).

$$G_P = \frac{T_1}{T_P}; \qquad G_P \leq P$$

7. Análisis del rendimiento

Podemos ver claramente a través de este gráfico de nuestro programa, que existe un aumento substancial del tiempo al ir aumentando la imagen. Tanto la imagen como el tiempo aumentan una cantidad más o menos similar de 4 veces del tamaño y el tiempo anterior.

Con lo cual, hay una relación directamente proporcional si aumentamos el parámetro x, el tamaño, sobre el parámetro y, el tiempo.

8. Parámetros de compilación

Sabemos por Arquitectura de los computadores (asignatura predecesora) que los programas reales pueden afectan al rendimiento de, en este, caso nuestra aplicación. Dentro de los programas reales podemos encontrar los compiladores y sus opciones de compilación, que pueden ayudar a mejorar el rendimiento de una aplicación.

Las prioridades de un compilador en el momento de traducir lenguajes de lato nivel son: en primer lugar la exactitud de la traducción, en segundo lugar la velocidad del código generado, en tercer lugar el tamaño del código, en cuarto lugar la velocidad del compilador, y en quinto lugar el soporte a la

depuración.

Tratando las optimizaciones a nivel de arquitecturas podemos ver las diferencias entre las arquitecturas CISC y RISQ teniendo en cuenta que cuanto más registros, mas fácil es optimizar el rendimiento, para lo que el compilador deben conocer los costes relativos entre las diversas instrucciones y elegir la mejor secuencia de instrucciones dado que CISC consta ofrece más alternativas, mientras RISC intenta limitarlas y conseguir una longitud constante en ellas.

Las ayudas más importantes que la arquitectura de un procesador puede prestar al diseño del compilador son: Regularidad del repertorio: simplificar la generación de código mediante la ortogonalidad de los elementos en él y proporcionar funciones primitivas.

A nivel de paralelismo podemos conseguir optimizaciones reordenando las operaciones para permitir que múltiples cálculos se produzcan en paralelo, ya sea en la instrucción, la memoria o a nivel de hilo.

Tratando el tema que nos ocupa hablaremos sobre las optimizaciones a nivel de parámetros en GCC.

Sin ninguna opción de optimización, el objetivo del compilador es el de reducir el coste de compilación y producir los resultados esperados. Muchas de las optimizaciones en GCC solo están habilitadas al introducir -0 en la línea de comandos (como si fuesen niveles). Dependiendo del “target” y de como GCC haya sido optimizado pueden activarse diferentes optimizaciones.

El objetivo de estas opciones, en general, es el de mejorar el espacio y tiempo mediante el sacrificio de otros aspectos, como el tiempo de compilación. Mas adelante trataremos una serie de parametros que nos pueden ser de ayuda en nuestra aplicación.

8.1 Parámetros de compilación usados

Hay diferentes tipos de optimizaciones, en general mejoran el espacio y tiempo, sacrificando otros aspectos. Algunas optimizaciones mejoran ambos aspectos a expensas de generar un fichero binario más largo o reduciendo el binario obtiendo un mayor tiempo de ejecución.
Por ejemplo, el espacio se puede optimizar "empaquetando" juntos los datos y funciones, y el tiempo se puede optimizar alineando los datos para satisfacer los requisitos de la arquitectura y hacer un mejor uso de la caché, pero se aumenta el espacio ocupado.

Otros ejemplos en los que se sacrifica el espacio en contra de un mejor tiempo son:

> **Loop unrolling**: usado en bucles pequeños, transformando estos a instrucciones individuales reduciendo / eliminando las comprobaciones en el bucle.
> **Function inlining**: en lugar de hacer una llamada a una función se "pega" el código de esta donde se haría la llamada, evitando saltos innecesarios en la pila.
> **Code reordening:** reordenar el código para ejecutar antes el código no dependiente de cálculos

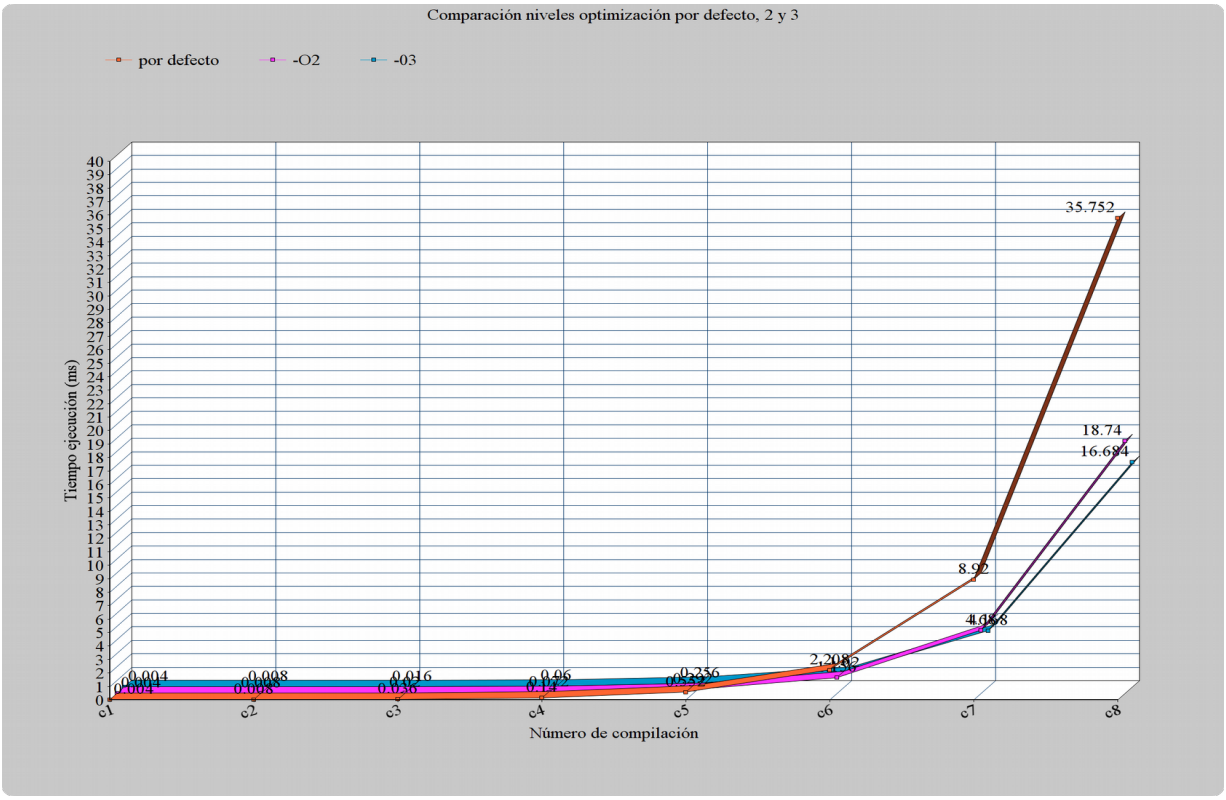
anteriores.

En cuanto al compilador g++ ofrece cuatro niveles de optimización:

O0: por defecto, reduce el tiempo de compilación, no da problemas al depurar.
O1: el compilador intenta reducir el tamaño del código, con pocas optimizaciones que mejoren el tiempo de ejecución
O2: Optimiza mucho más, mejorando tanto el espacio como el tiempo, a este se le incluyen odas las del nivel O1 además de otras nuevas.
O3: mejora aun más la optimización.

Cada uno de los niveles descritos, implican una bateria de flags tras de sí, siendo en general los de nivel superior la suma de los anteriores más otros nuevos.

En la siguiente gràfica se pueden ver los tiempos de ejecución del programa presentado en esta documentación, ejecutándolo en las máquinas de la eps:



Como se puede observar, en las tres línias, en las sucesivas compilaciones el tiempo de ejecución aumenta en gran cantidad tras llegar a la sexta compilación.

Además es notable la disminución del tiempo de ejecución en la última compilación usando los niveles de compilación por defecto del compilador g++.

Por último voy a explicar los unos de los flags escogidos, que major relación tienen con nuestro programa y una gràfica comparando con los nuevos resultados:

- **fipa-cp:** Esta optimización analiza el programa para determinar cuando los valores pasados a las funciones son constantes y luego se optimiza en consecuencia. Esta optimización puede mejorar sustancialmente el rendimiento cuando se tienen constantes pasadas a funciones.

Este flag serà de utilidad en la función de interpalción, a causa de la constantes llamadas a esta y la cantidad de parámetros pasados.

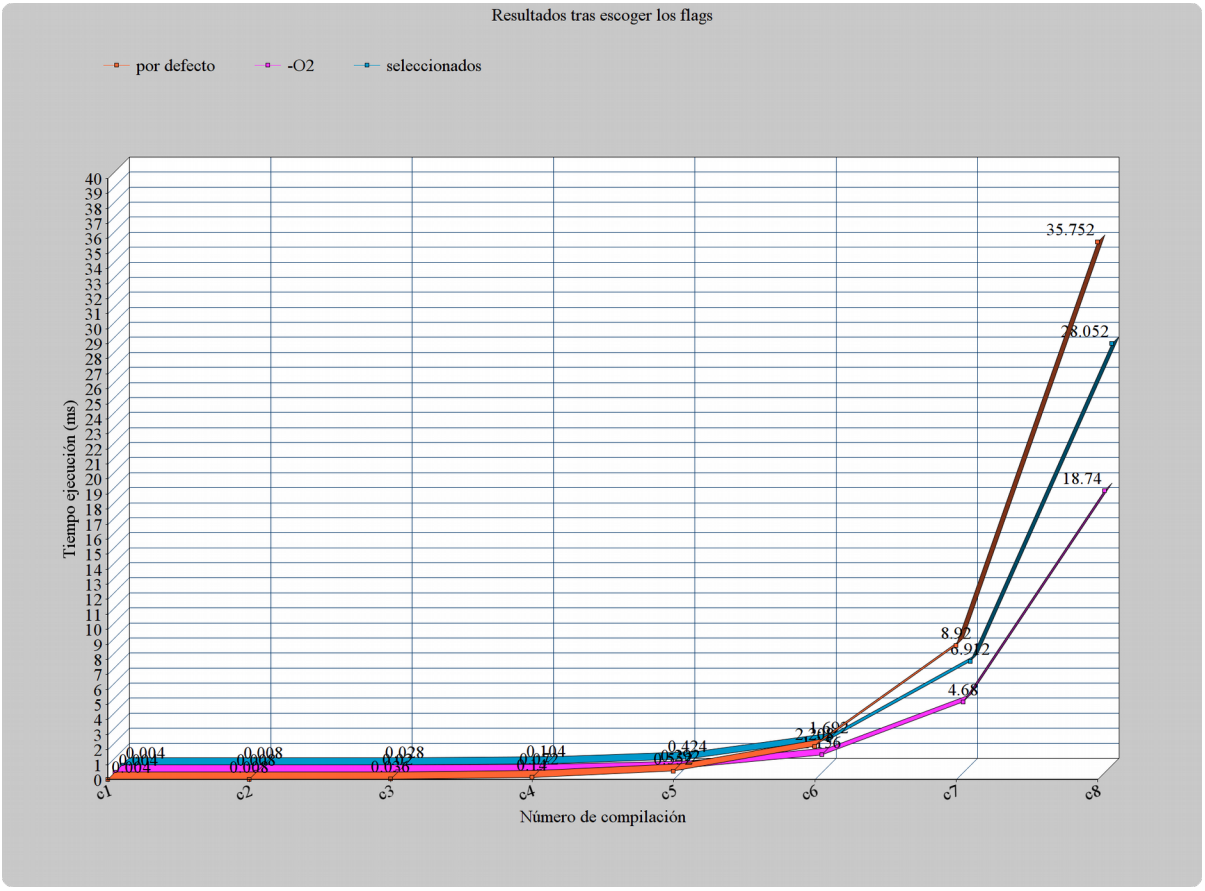
- **fpartial-inlining: Esta optimización hace parte de la funciones en linea.**

Este será aplicable tanto a la función de interpolación, como la usada para calcular el tiempo de ejecución, cuyos beneficiós han sido explicados anteriormente.

- fipa-icf: realiza un plegado de codigo repetido en funciones y variables. Esta optimizacion reduce el tamaño del codigo.

En la función principal, donde se recorre la información de la imagen y se hacen las llamadas a la función de interpalación, se realizan muchas veces la misma operación para cada uno de las iteraciones de los bucles.

Los resultados, comparados con los niveles de optimización de g++, tras usar los flags explicados son:



Como se puede observar, hay una disminución en casi 8 segundos tan solo usando los 3 flags comentados, no siendo equiparable al nivel 2, pero si es notòria teniendo en cuenta que este usa una gran cantidad de optimizaciones.