Práctica de programación paralela

Grupo 84

Equipo 6

Carlos Peña Martin (100405802)
Alvaro Morata Hontanaya (100405846)
Miguel Castuera Garcia (100451285)
Javier Moreno Yebenes (100428998)

Índice

Paralelización

Dado que el código secuencial nos lo suministraban ya, la parte de desarrollo de código era simplemente incluir la librería de OpenMP para poder paralelizar los métodos de transformación a escala de grises, generación del histograma y difusión mediante filtro de Gauss tanto para la estructura de arrays (SOA) como para el array de estructuras (AOS).

El código suministrado ya cumple con las normas de calidad reflejadas en el enunciado que debe tener.

Para la paralelización hemos utilizado los recursos del laboratorio 4 de programación paralela con OpenMP. Hemos incluido simplemente la **paralelización del primer bucle for** dentro de cada método (*to_gray()*, *generate_histogram()* y *gauss()*) de cada tipo de estructura.

La línea incluida justo antes de cada for ha sido la siguiente: **#pragma omp parallel for**

Esta línea sirve para que cada hilo coja un número de iteraciones determinadas y las ejecute de forma paralela a los demás. Este número de iteraciones varía según la política de planificación.

Cabe destacar que no se ha especificado en la línea los hilos y la planificación utilizada, puesto que eso se ha realizado con las siguientes variables de entorno en el script de ejecución ("*perf_avignon.sh*"):

- **OMP_NUM_THREADS:** corresponde al número de hilos que se utilizará para paralelizar las operaciones con OpenMP. El valor de esta variable se ha establecido a los valores pedidos en el enunciado: 1, 2, 4, 8 y 16.
- **OMP_SCHEDULE:** corresponde a la planificación utilizada en la ejecución. Se ha establecido a los valores "*static*", "*dynamic*" y "*guided*".

La política estática planifica bloques de iteraciones (de tamaño N) para cada hilo. En la dinámica cada hilo toma un bloque de N iteraciones de una cola hasta que se han procesado todas. Por último, con la guiada, cada hilo toma un bloque de iteraciones hasta que se han procesado todas. Se comienza con un tamaño de bloque grande y se va reduciendo hasta llegar a un tamaño N.

Dado que todos los bucles *for* paralelizados se ejecutan un número de iteraciones igual al número de píxeles de la imagen y la única imagen a procesar tiene un número de píxeles múltiplo de 32, todos los hilos terminan ejecutando el mismo número de iteraciones.

Analizamos también si era necesario pasar variables privadas a cada hilo en caso de que se necesitase que no fueran compartidas, pero vimos que no era necesario.
No se necesita más código para paralelizar, ya que si paralizamos otro bucle dentro de cada uno, se formarían muchos más hilos y no los que queremos.

Evaluación de Rendimiento y Energía

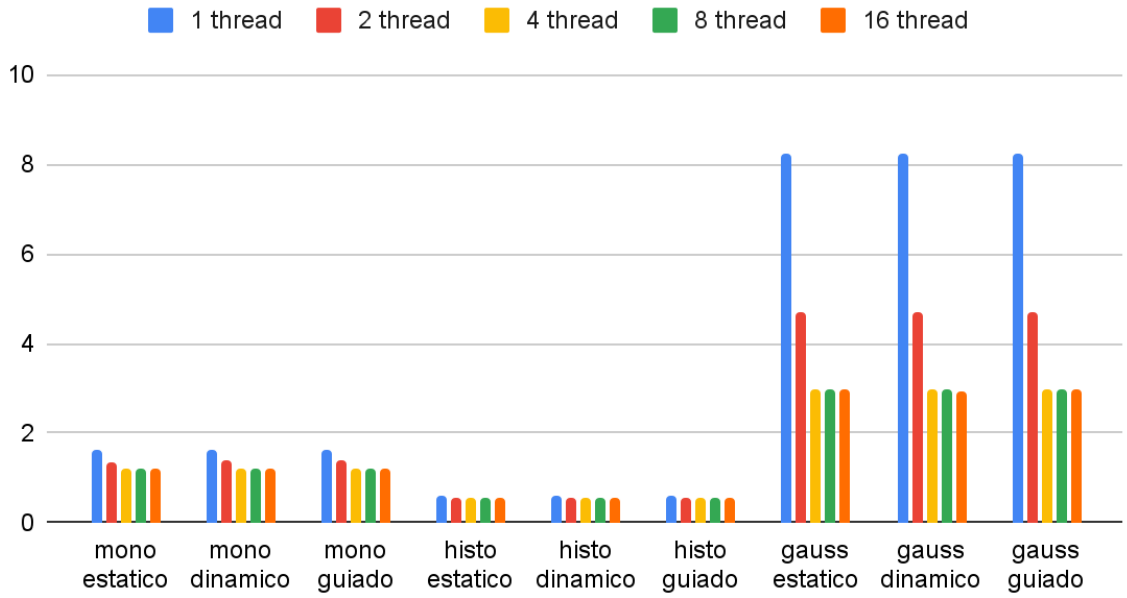
Se han realizado en total 90 pruebas: por cada tipo de estructura (soa y aos), por cada uno de los tres tipos de planificación de los hilos (estático, dinámico y guiado), por cada método (escala de grises, histograma y gauss) y por número de hilos utilizados (1, 2, 4, 8 y 16).

Todas las pruebas realizadas han sido aplicadas únicamente a la imagen BMP "sabatini.bmp" suministrada, teniendo un tamaño de 1.980x1.320 píxeles=2613600.

Evaluación de rendimiento

En este apartado analizaremos los tiempos de ejecución de cada método (escala de grises, histograma y gauss).

Aquí se muestran los tiempos de ejecución de structure o arrays (SOA):



SOA tiempos de ejecución

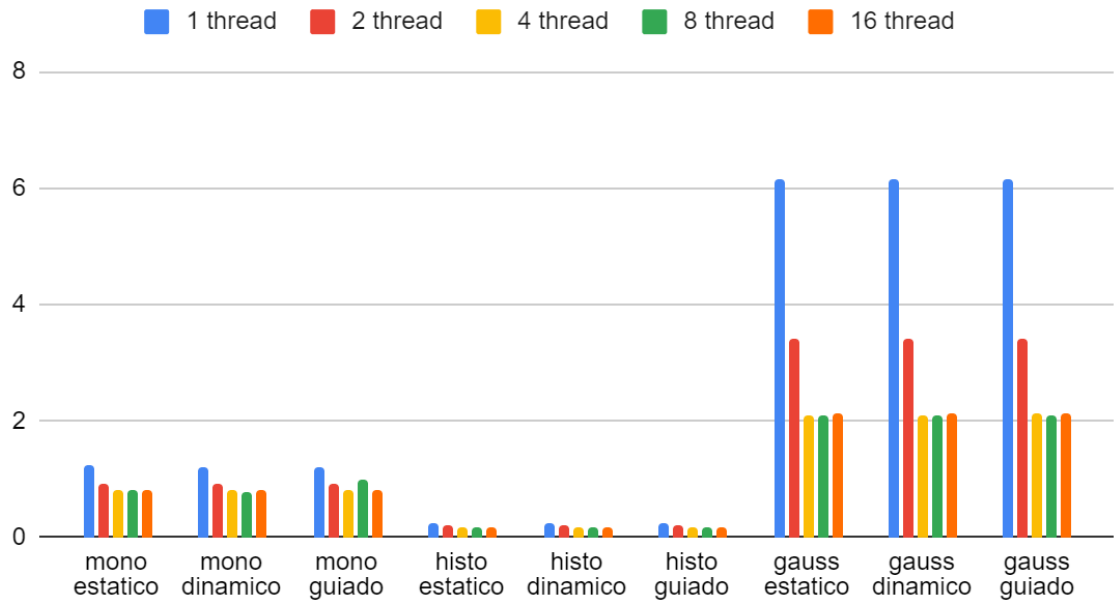
Vemos claramente la diferencia de tiempos entre los tipos de operaciones.

En el caso de la operación "*histo*" es donde encontramos los tiempos más bajos. Esto tiene sentido ya que en esta función sólo estamos extrayendo la información de los colores de cada imagen y no estamos aplicando nada encima de la misma.

En segundo lugar tenemos la operación de escala de grises, este dado que modifica los colores al aplicar el filtro tarda más.

Por último tenemos el filtro de "gauss" que es el que más gasta. Gauss, al tener un algoritmo compuesto por dos sumatorios, es más complejo que el algoritmo de mono. Es por esto que este último requiere de un menor tiempo de ejecución.

Aquí se muestran los tiempos de ejecución de arrays of structures (AOS):



AOS tiempos de ejecución

En los gráficos de barras anteriores podemos ver las ejecuciones de las distintas operaciones tanto del AOS como del SOA. A simple vista se puede observar como todas las ejecuciones del AOS son más eficientes que en el caso del SOA (tardan aproximadamente un 75% de lo que tardan las de SOA).

Este es un claro ejemplo de que, como vimos en clase, los array de estructuras mejoran el rendimiento respecto a las estructuras de arrays, debido a que al estar organizadas de manera diferente, los accesos a memoria caché de datos tienen menor número de fallos y, por lo tanto, menos penalizaciones de tiempo por fallos, ahorrando bastante tiempo.

Aun así, las operaciones siguen la misma tendencia en ambas, la que hemos descrito anteriormente.

Ahora analizaremos las diferencias de tiempo de ejecución entre hilos. Ya de primeras se puede observar que con solamente 1 hilo (sin paralelización) se tarda más que utilizando paralelización.

De las 5 diferentes cantidades de hilos utilizados, se puede ver que la mejor opción en general es con 4 hilos a la vez, esto es debido a que la arquitectura del servidor de avignon en el que se ha compilado y ejecutado tiene 4 núcleos y así la repartición es perfecta siendo 1 hilo por núcleo. A partir de ahí no mejora casi el tiempo de ejecución, llegando a ser incluso algo peor el utilizar 8 o 16 hilos en algunas ocasiones.

Analizando por tipo de operación, podemos observar que en la operación de gauss con 2 hilos se reduce aproximadamente a la mitad el tiempo y con 4 se reduce otra vez a la mitad más o menos. Por otro lado, en las operaciones de mono e histograma disminuye mucho menos el tiempo de ejecución, esto se debe a que, como hemos explicado antes, el filtro de

gauss realiza más operaciones en el bucle que el de mono, por lo tanto el speedup de la paralelización del método de escala de grises resulta menor.

Por último, el speedup de la paralelización de la operación del histograma es casi nulo, ya que, al igual que hemos dicho antes, éste método sólo extrae la información de los colores de cada imagen y no modifica los píxeles.

Por otra parte, con 8 hilos y 16 hilos también se obtiene una buena optimización ya que son múltiplos de 4 y cada núcleo tiene así el mismo número de hilos y las tareas están perfectamente repartidas, aunque el tener que cambiar de hilo más veces hace que gaste más tiempo, añadiendo ese tiempo de cambio entre hilos.

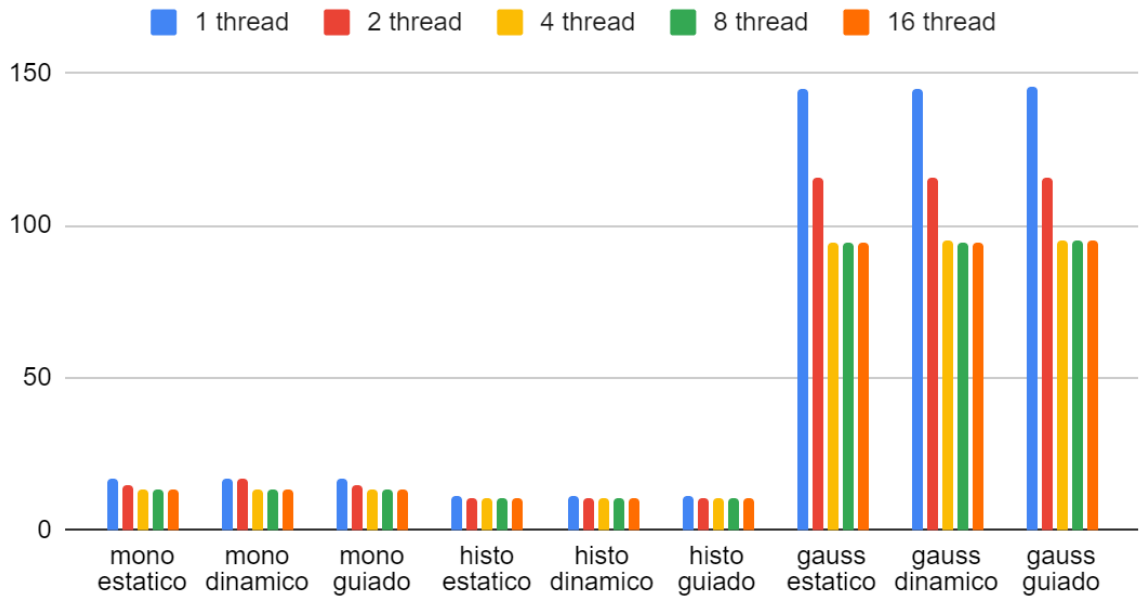
Analizando las políticas de planificación vemos que no hay casi diferencia entre los tiempos de ejecución de las tres diferentes políticas utilizadas. Aunque la poca mejora se produce con la planificación dynamic, casi estando a la par con guided. Esto es debido a que, al planificar estáticamente, cada hilo en un principio coge una división exacta de las iteraciones. Por otro lado, con dynamic y guided van cogiendo poco a poco tareas de la cola por lo que se adapta al momento y es más rápido.

Dado que el número de píxeles de la imagen es múltiplo de 16, y por tanto de 8, 4 y 2 también, la división de iteraciones entre hilos siempre va a ser la misma en cada planificación, siendo por ello los tiempos de ejecución prácticamente iguales.

Evaluación de energía consumida

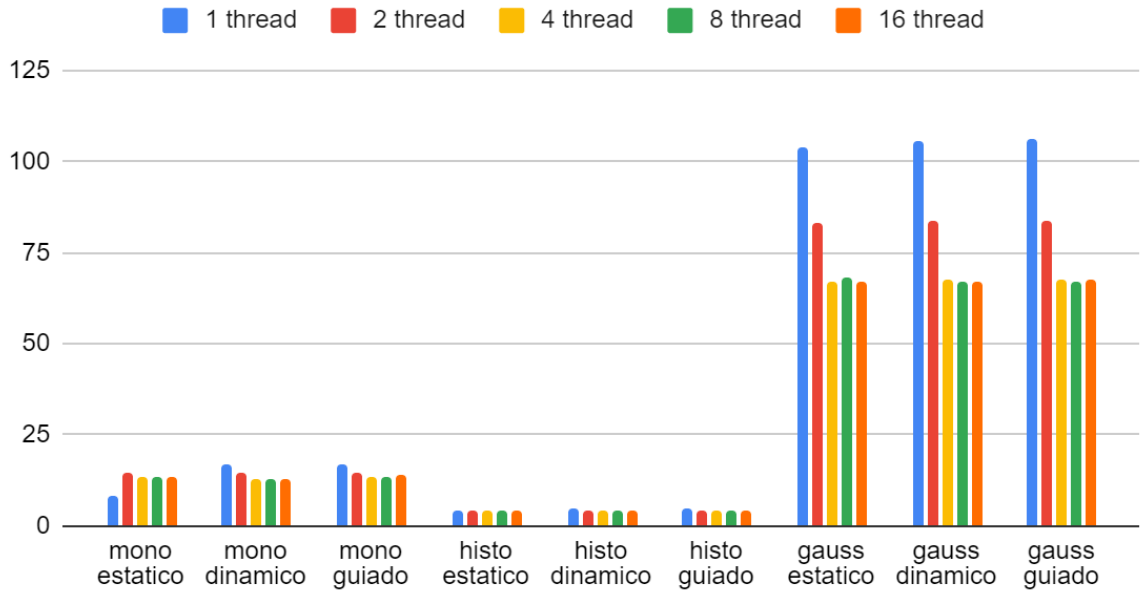
En este apartado analizaremos la energía consumida, medida en Julios, de cada método (escala de grises, histograma y gauss).

Aquí se muestra la energía consumida de structure o arrays (SOA):



SOA consumo energético

Aquí se muestran la energía consumida de arrays of structures (AOS):



AOS consumo energético

Comparando a rasgos generales la ejecución del AOS y del SOA podemos ver que siguen la misma tendencia incluso a nivel de hilo. Aun siendo similares, la diferencia más notable entre ambas es la diferencia de energía consumida. Esta diferencia siempre es superior en el caso del AOS. A continuación veremos en más detalle cada ejecución y compararemos entre ambos modos.

Empezaremos por la función que menos energía requiere, la cual es "*histo*". Esto se debe a que es la que menos cálculos realiza para completarse, por lo tanto, requiere menos energía. Como se puede apreciar en los gráficos, la diferencia entre las políticas de planificación y el uso de múltiples hilos no es apenas relevante a la hora de reducir el consumo ya que los valores para todos ellas son prácticamente los mismos. Eso sí, la diferencia entre el modo AOS y SOA es notable. Los valores para el array of structures duplican el consumo en todas las ejecuciones.

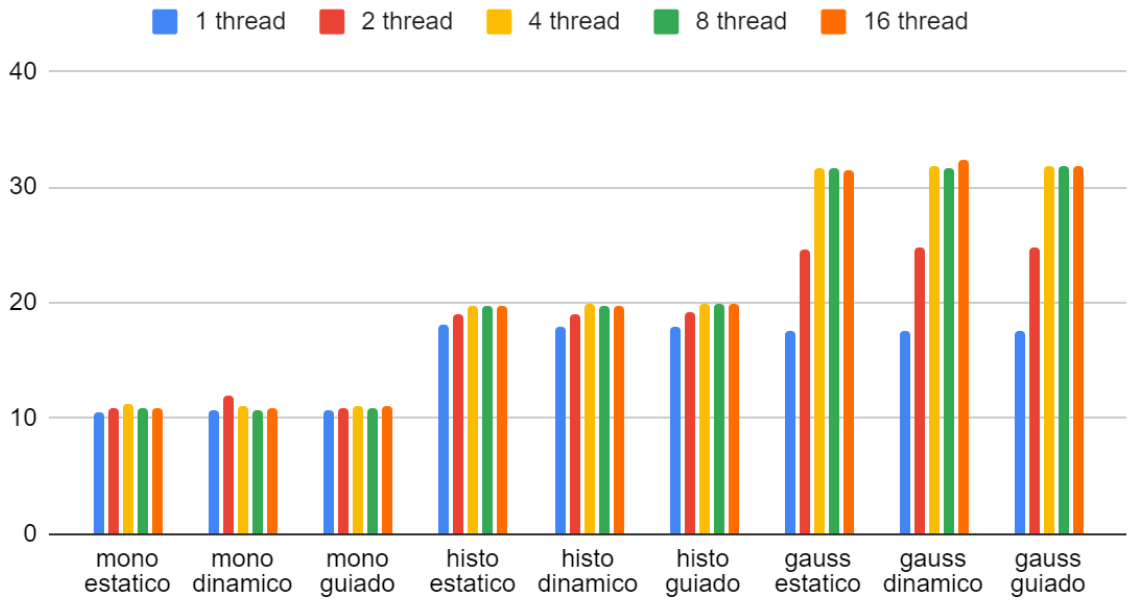
Continuaremos con la función "*mono*", la cual requiere más energía que la mencionada previamente. En este caso se vuelve a consumir más energía en el modo AOS que en el SOA y las políticas de planificación vuelven a influir de forma similar. A diferencia de la anterior función, el consumo si que tiene cierta dependencia del número de hilos. La diferencia entre usar un hilo o de dos en adelante es notable, siendo el primer caso el más costoso comparado con los otros hilos que lo van reduciendo progresivamente. Eso sí, una vez el número de hilos es superior a 4, el valor se mantendrá o se incrementará por lo que este es el valor más óptimo.

Finalmente, hablaremos sobre la función que más energía consume. Esta función es "*gauss*" ya que es la que más operaciones y aplicaciones realiza de todas. Como se puede observar, vuelve a ocurrir lo mismo que en la función "*mono*". El Array Of Structures consume bastante más que la Structure Of Arrays, la política de planificación no influye notablemente y la influencia de los hilos en el consumo está en la misma línea que en "*mono*".

Evaluación de potencia utilizada

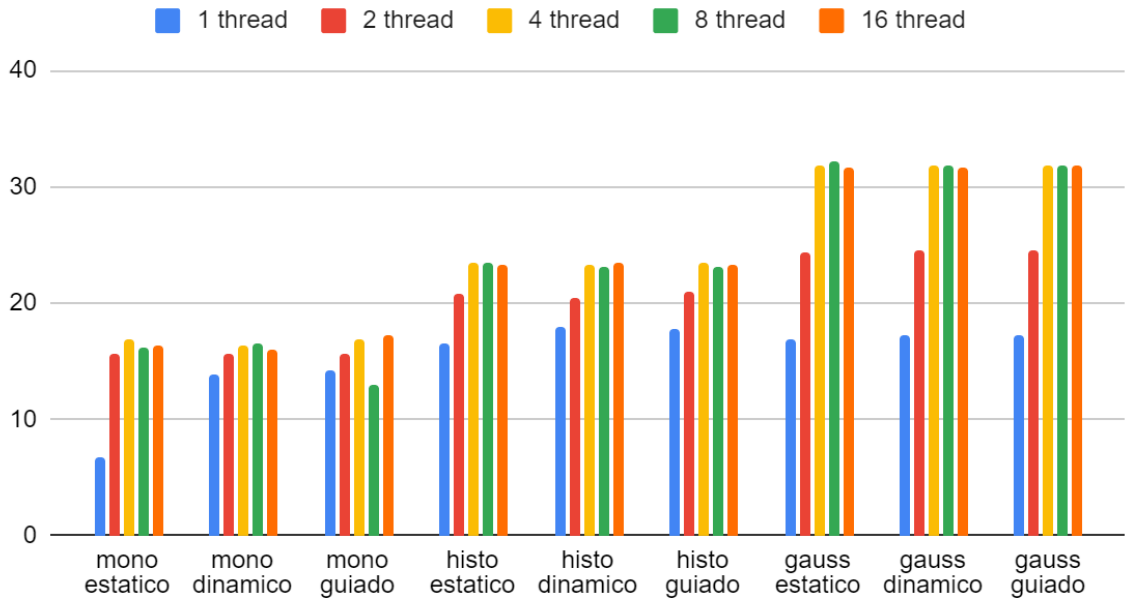
En este apartado analizaremos la potencia utilizada de cada método (escala de grises, histograma y gauss).

Aquí se muestra la potencia utilizada de structure o arrays (SOA):



SOA uso de potencia

Aquí se muestran la potencia utilizada de arrays of structures (AOS):



AOS uso de potencia

Comparando a rasgos generales la ejecución del AOS y del SOA podemos ver que nuevamente siguen la misma tendencia. En este caso, la diferencia más notable entre ambas es que el AOS requiere, en líneas generales, mayor potencia en comparación al SOA. A continuación veremos en más detalle cada ejecución y compararemos entre ambos modos.

Comenzaremos con la función “*mono*”, la cual usa una menor potencia en ambos modos. En las gráficas podemos observar que el caso de array de estructuras consume casi el doble en todos los casos. Respecto a los hilos, todos consumen más o menos los mismo exceptuando en el AOS con un único hilo, el cual consume bastante menos que el resto. En cuanto a la política de planificación, en el SOA no afecta visiblemente mientras que en el AOS si que varía un poco los valores, más notablemente en los que usan 1 y 8 hilos. Aun así estas variaciones son muy leves y apenas influyen en el resultado final.

En cuanto a la función “*histo*”, en esta nuevamente la potencia necesaria es mayor en el AOS respecto del SOA. Además la política de planificación vuelve a dar los mismos valores sin importar la que se use. Por el contrario, en esta función si que se puede ver cierta tendencia dentro de los hilos. En este caso, contra menos hilos use, menos potencia necesitará. Eso sí, cuando haya 4 hilos o más, la potencia necesaria será más o menos la misma.

Finalmente, la función que más potencia necesita es “*gauss*”. Esta función sigue el mismo patrón que la anterior. Es decir, necesita más potencia en el AOS que en el SOA, las políticas de planificación dan los mismos resultados y contra menos hilos, menos potencia hasta que se usan 4 o un número superior.

Organización del Trabajo

1. **Diseño del código (Paralelización)**
   Paralelización de los bucles de los métodos - Carlos Peña 1h

2. **Compilación y ejecución**
   Creación del fichero de compilación y del script de ejecución - Álvaro Morata 3h
   Ejecución de todas las pruebas realizadas - Álvaro Morata 2h

3. **Recopilación y procesado de la información**
   Traspaso de los resultados a tablas - Miguel Castuera 2h
   Generación de gráficos a partir de tablas - Miguel Castuera 2h

4. **Memoria**
   Paralelización - Carlos Peña 1h
   Evaluación del rendimiento - Carlos Peña 3h
   Evaluación de la energía y potencia - Javier Moreno 3h
   Organización del trabajo - Carlos Peña 0,5h
   Conclusiones - Javier Moreno 1,5h
   Anexo: Scripts creados - Álvaro Morata 0,5h

Conclusiones

Finalmente, para concluir la memoria vamos a hablar sobre los resultados obtenidos y mencionaremos algunos comentarios acerca de nuestra experiencia con la práctica.

En esta práctica hemos aprendido cómo paralelizar la ejecución de un programa en un número determinado de hilos. Gracias a esto y a los resultados obtenidos, hemos podido ver cómo afectan realmente al rendimiento del mismo. Como hemos visto anteriormente, un mayor número de hilos proporciona un mayor rendimiento en líneas generales, pero no siempre. Por ello, hemos sido capaces de ver en qué punto deja de ser útil este método además de ver las razones de ello.

En comparación a la anterior práctica, esta nos ha resultado bastante más asequible a niveles de tiempo disponible para hacerla y de dificultad.

Anexo: Scripts creados

Como se puede observar en los archivos entregados en el apartado de código, se ha decidido entregar los scripts, tanto los de compilación como los de ejecución, por si fuesen útiles para el profesorado a la hora de corregir la práctica. A continuación se incluye una breve descripción de cada uno de los scripts:

- **build_avignon.sh**
  Se encarga de ejecutar cmake con la versión 12.1.0 de GCC. Además, copia el script "*make_script.sh*" a la carpeta *release*, en la que estarán los binarios una vez se compilen los archivos fuente.
- **make_script.sh**
  Se encarga simplemente de ejecutar *make* y compilar el programa. Este fichero hay que ejecutarlo sobre el directorio *release*.
- **perf_avignon.sh**
  Se encarga de ejecutar todas las pruebas con *perf* y cambiar constantemente el valor de las variables de entorno *OMP_NUM_THREADS* y *OMP_SCHEDULE* para realizar las pruebas pertinentes. Se ha intentado dentro de lo posible ofrecer una legibilidad lo más alta posible a este archivo.