15 DE ABRIL DE 2016

MULTIPLICACIÓN DE MATRICES

FRANCISCO JESÚS ARCOS ROMERO
i32arrof@uco.es

Tabla de Contenidos

1. INTRODUCCIÓN

El trabajo consiste en una comparación de los tiempos de procesamiento al realizar la multiplicación de dos matrices cuadradas mediante un programa secuencial usando para ello Pthread y OpenMP. Planteando en cada uno de ellos qué bucle, en este caso de tipo for. ¿qué bucle es mejor usar? ¿Qué programación paralela es mejor? ¿Con cuantos hilos se trabaja mejor en esta computador? Son algunos de los objetivos cumplir con este trabajo.

2. DESCRIPCIÓN DEL PROCESADOR

A la hora de realizar el trabajo se usa una computadora que contiene un procesador Procesador Intel® Core™ i5-2430M siendo la velocidad del procesador 2,40 GHz. Contiene 2 núcleos físicos y 4 núcleos lógicos.

La memoria caché es una memoria pequeña y rápida que se interpone entre la CPU y la memoria principal para que el conjunto opere a mayor velocidad. Para ello es necesario mantener en la caché aquellas zonas de la memoria principal con mayor probabilidad de ser referenciadas. Esto es posible gracias a la propiedad de localidad de referencia de los programas.

La estructura del sistema de memoria Caché por núcleo es la siguiente:

Tipo de Cache	Tamaño	Asociatividad	Tamaño de Palabra	Tipo de Mapeo
L1 Data	32KB	8 conjuntos	64B	Directo
L1 Instrucciones	32KB	8 conjuntos	64B	Directo
L2 Unificada	256KB	8 conjuntos	64B	Directo
L3 Unificada	3MB	12 conjuntos	64B	Complejo*

* Al tener 3MB no se trata de potencia de dos por lo que tendrá algún tipo de complejidad para el mapeo.

Asociativa por conjuntos: Cada bloque de la memoria principal tiene asignado un conjunto de la caché, pero se puede ubicar en cualquiera de los bloques que pertenecen a dicho conjunto. Ello permite mayor flexibilidad que la correspondencia directa y menor cantidad de comparaciones que la totalmente asociativa.

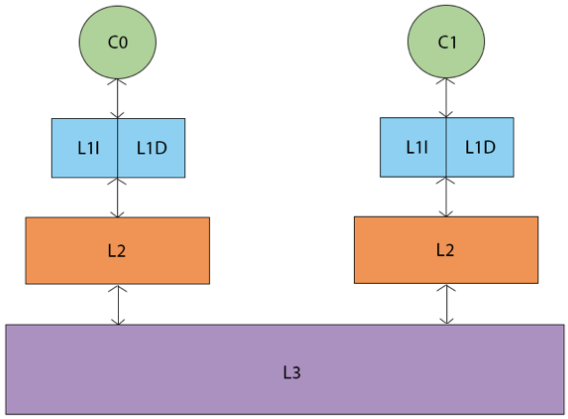
2.1. LOCALIDAD DE REFERENCIA: TEMPORAL Y ESPACIAL

Los programas manifiestan una propiedad que se explota en el diseño del sistema de gestión de memoria de los computadores en general y de la memoria caché en particular, la *localidad de referencias*: los programas tienden a reutilizar los datos e instrucciones que utilizaron recientemente. Una consecuencia de la localidad de referencia es que se puede predecir con razonable precisión las instrucciones y datos que el programa utilizará en el futuro cercano a partir del conocimiento de los accesos a memoria realizados en el pasado reciente. La localidad de referencia se manifiesta en una doble dimensión: temporal y espacial.

Localidad temporal: las palabras de memoria accedidas recientemente tienen una alta probabilidad de volver a ser accedidas en el futuro cercano. La localidad temporal de los programas viene motivada principalmente por la existencia de bucles.

Localidad espacial: las palabras próximas en el espacio de memoria a las recientemente referenciadas tienen una alta probabilidad de ser también referenciadas en el futuro cercano. Es decir, que las palabras próximas en memoria tienden a ser referenciadas juntas en el tiempo.

2.2. ESTRUCTURA DE LA MEMORIA CACHÉ DEL SISTEMA



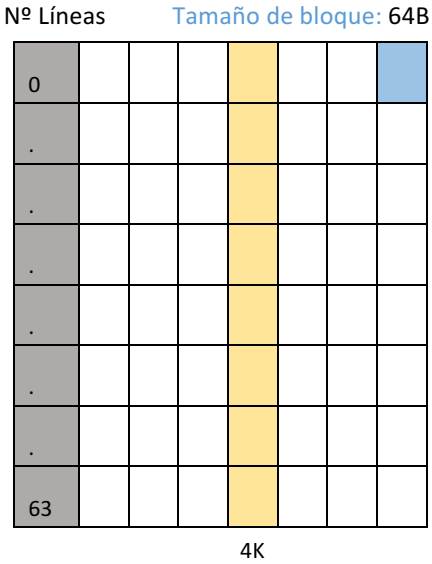
Estructura de la Memoria Caché del procesador

Memoria Caché L1

La caché almacena 32768 palabras de memoria principal. Como tiene 8 vías, almacena 32768 /8 = 4096 palabras por cada vía. Teniendo en cuenta que cada bloque tiene 64 palabras, existirán 4096 /64 = 64 bloques en cada vía, por lo que la caché tendrá 64 líneas, y necesitará $\log_2(64) = \log_2 2^6$ = 6 bits para seleccionar la línea de caché. Para seleccionar una palabra dentro del bloque se necesitarán $\log_2(64) = \log_2 2^6$= 6 bits.

Formato de la Memoria Caché L1

Línea	Palabra
6 Bits	6 Bits

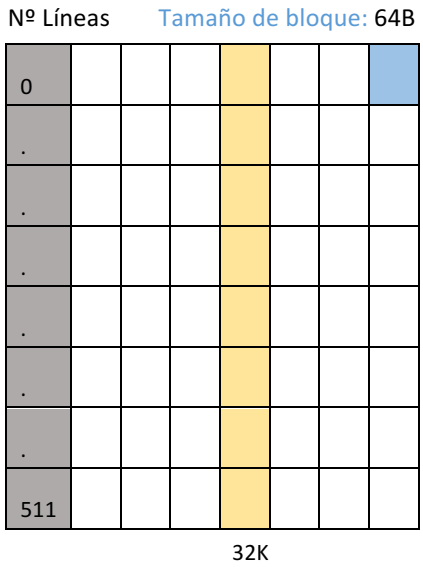


Memoria Caché L2

La caché almacena 262144 palabras de memoria principal. Como tiene 8 vías, almacena 262144/8 = 32768 palabras por cada vía. Teniendo en cuenta que cada bloque tiene 64 palabras, existirán 32768/64 = 512 bloques en cada vía, por lo que la caché tendrá 512 líneas, y necesitará $\log_2(512) = \log_2 2^9$ = 9 bits para seleccionar la línea de caché. Para seleccionar una palabra dentro del bloque se necesitarán $\log_2(64) = \log_2 2^6$= 6 bits.

Formato de la Memoria Caché L2

Línea	**Palabra**
9 Bits	6 Bits



2.3. MECANISMO DE ACTUALIZACIÓN DE LA CACHÉ

Para implementar el mecanismo de actualización de la caché con los datos con mayor probabilidad de ser refernciados se divide la memoria principal en bloques de un número de bytes (4,8,16 etc.) y la caché en marcos de bloque o líneas de igual tamaño. El bloque será, pues, la unidad de intercambio de información entre la memoria principal y la caché, mientras que entre la caché y la CPU sigue siendo la palabra. El *directorio* contiene la información de qué bloques de *Memoria principal* se encuentran ubicados en *Memoria caché.*

3. PREPARACIÓN PARA EL PROGRAMA

Se han creado las siguientes funciones para la generación, relleno y liberación de memoria de las matrices:

int ** reservaMemoria(int ** m, int numCol, int numFil);

int ** rellenaMatriz(int ** m, int numCol, int numFil);

int ** rellenaMatrizCero(int ** m, int numCol, int numFil);

int ** liberarMatriz (int ** m, int numFil, int numCol);

Estas serán ejecutadas antes y después de las operaciones para calcular su tiempo de ejecución.

La función rellenaMatrizCero () inicializa una matriz de forma dinámica a 0 para ser usada como matriz resultado.

Las variables numFil y numCol indican el tamaño de la matriz, al tratarse de una matriz cuadrada serán del mismo valor.

4. PROGRAMA SECUENCIAL

Se genera una Matriz de 768x768, es decir una matriz cuadrada.

El número del tamaño de matriz que se toma debe ser múltiplo del número de hilos. Si la matriz no fuera de un tamaño divisible por el número de hilos, el último bloque de memoria usada para almacenar los datos de una columna de la matriz también tendría información de la siguiente columna, con lo que a la hora de hacer una iteración se leería esa posición de memoria varias veces innecesariamente. Para evitar esta situación se tendría que añadir un relleno para conseguir ese tamaño.

El código de este programa se encuentra en el fichero: multiplicarMatriz.c

4.1. BUCLE I-J-K

Usando el siguiente bucle:

```
for(i=0; i<numFil; i++){
        for(j=0; j< numCol; j++){
                for(k=0; k< numCol; k++){
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];
                }
        }
}
```

Se consigue un tiempo de ejecución de 4.761149 segundos, sería la ejecución normal del programa en el que el bucle mas externo es i y el mas interno es k. En la matriz A se aprovecha la localizad espacial de la cache mientras en que la matriz B tiene que ir saltado posiciones de memoria. Este bucle aprovecha mas la memoria cache con respecto a la localidad espacial ya que funciona igual que la estructura de la memoria.

4.2. BUCLE I-K-J

Usando el siguiente bucle:

```
for(i=0; i<numFil; i++){
        for(k=0; k< numCol; k++){
                for(j=0; j< numCol; j++){
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];
                }
        }
}
```

Se consigue un tiempo de ejecución de 2.768531 segundos ya que se está accediendo a posiciones consecutivas de memoria caché aprovechando la localidad espacial y temporal al máximo lo que hace que la lectura sea mucho mas rápida.

4.3. BUCLE J-K-I

Usando el siguiente bucle:

```
for(j=0; j<numFil; j++){
        for(k=0; k< numCol; k++){
                for(i=0; i< numCol; i++){
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];
                }
        }
}
```

Se consigue un tiempo de ejecución de 6.624201 segundos al no estar operando con posiciones consecutivas de memoria y al estar operando sobre el bucle mas externo j e i el mas interno tiene que ir comprobando fila y columna luego pasa a otra fila misma columna y así hasta el final hace que el tiempo de ejecución sea mucho mas lento que las demás opciones, con lo que no se usa nada la localidad espacial ni temporal de la memoria cache al tener que ir leyendo y grabando datos continuamente.

5. PROGRAMA PARALELO USANDO PTHREAD

El código de este programa se encuentra en el fichero: multiplicarMatrizHilos.c

Al igual que en programa secuencial, se genera una matriz de 768x768 pero a parte se debe crear las correspondientes líneas relacionadas con los hilos:

Crea los hilos llamando a la función multiplicar, recibiendo el hilo correspondiente con: pthread_create(&hilos[h], NULL, multiplicar, (void *) &v[h]).

Recoge el hilo al finalizar con su cometido y finaliza el hilo con:

pthread_join (hilos[h], NULL).

Se ha tenido que crear una nueva función llamada multiplicar, que recibe un argumento que será el hilo que ejecuta la función. Gracias a saber cual es el hilo se crea la variable *intervalo*

que al dividirla por numero de filas dará el tamaño que se le asignará para cada hilo. Este tamaño (intervalo) se multiplica por el número de hilos y resolverá la posición por la que va a comenzar el hilo. A este resultado se le suma el intervalo dará la posición donde deberá finalizar el hilo.

```
void * multiplicar (void * num){
int inicio, fin, intervalo;
int i,j,k,m=0;
int * numhilo=(int*)num;
intervalo=numFil/numHilos;
inicio=*numhilo*intervalo;
fin=(inicio+intervalo);
for(i=inicio;i<fin;i++){
        for (j=0;j<numCol;j++){
                for (k=0;k<numCol;k++){
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];
                }
        }
}
pthread_exit(NULL);
}
```

Con *pthread_exit(NULL);* se realiza un retorno del hilo que esté en ejecución. En este caso se devuelve *NULL* al no tener que devolver ningún valor que tenga que recoger la función *pthread_join (hilos[h], NULL).*

Se comprueba el programa modificando el orden de las variables del bucle pero tal como se ha explicado en el programa secuencial, aquí vuelve a ocurrir lo mismo dando como mejor solución el bucle i k j ya que aprovecha más la localidad espacial y temporal de la memoria caché. Los resultados son los siguientes según el número de hilos:

5.1. PARALELIZACIÓN CON 2 HILOS

Número de Hilos = 2

Bucle i k j = 1.607356071 segundos

5.2. PARALELIZACIÓN CON 4 HILOS

Número de Hilos = 4

Bucle i k j = 1.473448038 segundos

5.3. PARALELIZACIÓN CON 8 HILOS

Número de Hilos = 8

Bucle i k j = 1.436833858 segundos

5.4. PARALELIZACIÓN CON 16 HILOS

Número de Hilos = 16

Bucle i k j = 1.444459915 segundos

5.5. CONCLUSIÓN DE PTHREAD

Una vez obtenidos los resultados mostrados anteriormente, se observa que a partir de 8 hilos el tiempo de ejecución se mantiene. Aunque el procesador sólo es capaz de crear 4 hilos virtuales a la vez, al crear 8 hilos se aprovecha la localidad espacial y temporal de la memoria caché, por lo tanto cuando finaliza un hilo y comienza otro se aprovecha que los datos están contiguos en memoria. Dando una mayor resultado en tiempo de ejecución.

Al usar más de 8 hilos se pierde un poco esta localidad de referencia puesto que no se sabe con exactitud que hilo está entrando antes, por lo que genera más fallo de caché.

6. PROGRAMA PARALELO USANDO OPENMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join. Está disponible en muchas arquitecturas, incluidas las plataformas de Unix y de Microsoft Windows. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influyen el comportamiento en tiempo de ejecución.

Definido conjuntamente por proveedores de hardware y de software, OpenMP es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas, para plataformas que van desde las computadoras de escritorio hasta supercomputadoras. Una aplicación construida con un modelo de programación paralela híbrido se puede ejecutar en un cluster de computadoras utilizando OpenMP y MPI, o a través de las extensiones de OpenMP para los sistemas de memoria distribuida.

Para iniciar e incluir OpenMP en nuestro código se debe de incluir la librería <omp.h> en el código del programa.

La primera directiva que se añade al programa es *omp_set_num_threads()*, a la que se le pasa como argumento el número de hilos que se va a usar para la ejecución, en la siguientes ejecuciones se usarán 2, 4, 8 y 16 hilos.

El siguiente código del programa se encuentra en el fichero: multiplicarMatrizOPM.c

6.1. DIRECTIVA *#PRAGMA OMP PARALLEL SHARED(A,B,C) PRIVATE(I,J,K){}*

Con *#pragma omp parallel shared(A,B,C) private(i,j,k){}*, estamos indicando donde comienza la paralelización del programa hasta que se cierre la llave. Cada hebra tiene su propio contexto de ejecución. Por defecto, todas son compartidas, salvo el iterador del bucle.

shared: Los datos de la región paralela son compartidos, lo que significa que son visibles y accesibles por todos los hilos. Por definición, todas las variables que trabajan en la región paralela son compartidas excepto el contador de iteraciones.

private: Los datos de la región paralela son privados para cada hilo, lo que significa que cada hilo tendrá una copia local que la usará como variable temporal. Una variable privada no es inicializada y tampoco se mantiene fuera de la región paralela. Por definición, el contador de iteraciones en OpenMP es privado.

A continuación vemos un ejemplo con dos hilos del resultado de realizar dicha paralelización.

```
#pragma omp parallel shared(A,B,C) private(i,j,k)
{
	for(i=0; i <numFil; i++){
		for(j=0; j<numCol; j++){
			for(k=0; k<numCol; k++){
				C[i][j] = C[i][j] + A[i][k] * B[k][j];
			}
		}
	}
}
```

Bucle i j k = 5.765455961 segundos

Bucle i k j = 3.663794994 segundos

Bucle j k i = 7.322361946 segundos

Se observa que los tiempos de ejecución son peores que los de secuencial y usando Pthread.

6.2. DIRECTIVA *#PRAGMA OMP FOR SCHEDULE (STATIC)*

En el siguiente caso se va a añadir la directiva *#pragma omp for schedule (static).* Con *for* se indica que se va a usar un bucle para la parelización con lo que la Hebra maestra crea hebras adicionales para cubrir las iteraciones del bucle.

schedule(tipo, chunk): Esto és útil si la carga procesal es un bucle do o un bucle for. Las iteraciones son asignadas a los hilos basándose en el método definido en la cláusula. Los tres tipos de scheduling son:

1 ***static***: Aquí todas las iteraciones se reparten entre los threads antes de que estos ejecuten el bucle. Se reparten las iteraciones contiguas equitativamente entre todos los threads. Especificando un integer como parámetro de *chunk* serán repartidas tantas iteraciones contiguas al mismo thread como el chunk indique.

2 ***dynamic***: Aquí al igual que en static se reparten todas las iteraciones entre los threads. cuando un thread en concreto acaba las iteraciones asignadas, entonces ejecuta una de las iteraciones que estaban por ejecutar. El parámetro *chunk* define el número de iteraciones contiguas que cada thread ejecutará a la vez.

3 ***guided***: Un gran número de iteraciones contiguas son asignadas a un thread. Dicha cantidad de iteraciones decrece exponencialmente con cada nueva asignación hasta un mínimo especificado por el parámetro *chunk*

En este caso nuestra opción es *static* al tratarse de matrices cuadradas mostrando el siguiente código:

```
#pragma omp parallel shared(A,B,C) private(i,j,k) {
        #pragma omp for schedule (static)
        for(i=0; i <numFil; i++){
                for(j=0; j<numCol; j++){
                        for(k=0; k<numCol; k++){
                                C[i][j] = C[i][j] + A[i][k] * B[k][j];
                        }
                }
        }
}
```

Número de Hilos = 2

Bucle i j k = 2.847526073 segundos

Bucle i k j = 1.832470894 segundos

Bucle j k i = 3.585833073 segundos

Se observa la reducción del tiempo casi a la mitad de los obtenidos anteriormente ya que se reparten de forma equitativa las matrices. Si la matriz no fuera cuadrada tendría que usarse la opción *dynamic* en vez de *static*.

A continuación todas las pruebas se van a realizar las pruebas con dos hilos y variando la posición de *#pragma omp for schedule (static)* sobre el bucle i k j el cual es el que está dando los mejores resultados. Una vez se obtenga el mejor resultado se pasará a modificar el número de hilos.

6.3. PARALELIZACIÓN DEL BUCLE I K J

```
#pragma omp parallel shared(A,B,C) private(i,j,k)
{
	#pragma omp for schedule (static)
	for(i=0; i <numFil; i++){
		for(k=0; k<numCol; k++){
			for(j=0; j<numCol; j++){
				C[i][j] = C[i][j] + A[i][k] * B[k][j];
			}
		}
	}
}
```

Número de Hilos = 2

Bucle i k j = 1.725558996 segundos

6.4. PARALELIZACIÓN DEL BUCLE K J

```
#pragma omp parallel shared(A,B,C) private(i,j,k)
{
	for(i=0; i <numFil; i++){
		#pragma omp for schedule (static)
		for(k=0; k<numCol; k++){
			for(j=0; j<numCol; j++){
				C[i][j] = C[i][j] + A[i][k] * B[k][j];
			}
		}
	}
}
```

Número de Hilos = 2

Bucle i k j = 2.099480152 segundos

6.5. PARALELIZACIÓN DEL BUCLE J

```
#pragma omp parallel shared(A,B,C) private(i,j,k)
{
	for(i=0; i <numFil; i++){
		for(k=0; k<numCol; k++){
			#pragma omp for schedule (static)
			for(j=0; j<numCol; j++){
				C[i][j] = C[i][j] + A[i][k] * B[k][j];
			}
		}
	}
```

}

Número de Hilos = 2

Bucle i k j = 7.070813894 segundos

6.6. CONCLUSIÓN SOBRE *#PRAGMA OMP FOR SCHEDULE (STATIC)*

Moviendo *#pragma omp for schedule (static)* por los distintos bucles lo que se consigue e paralelizar el los bucles que haya por debajo de la directiva. Se consigue un mejor rendimiento en paralelizando i-k-j. Esto se debe a que los hilos se reparten equitativamente dentro del bucle for al completo con las tres variables con lo que consigue un mayor rendimiento colocando la directiva en esa posición.

6.7. PARALELIZACIÓN CON 2 HILOS

Número de Hilos = 2

Bucle i k j = 1.730787039 segundos

6.8. PARALELIZACIÓN CON 4 HILOS

Número de Hilos = 4

Bucle i k j = 1.611748934 segundos

6.9. PARALELIZACIÓN CON 8 HILOS

Número de Hilos = 8

Bucle i k j = 1.547828913 segundos

6.10. PARALELIZACIÓN CON 16 HILOS

Número de Hilos = 16

Bucle i k j = 1.573575974 segundos

6.11. CONCLUSIÓN DE OPENMP

Al igual que ocurre con Pthread, Se observa que a partir de 4 hilos el tiempo de ejecución se mantiene y esto se debe a que el procesador solo es capaz de crear 4 hilos virtuales a la vez con lo que no se va a conseguir una mejora de tiempo de ejecución, al menos con este procesador.

7. COMPARACIÓN SECUENCIAL CON PARALELO

En este apartado se propone la comparación entre el programa secuencial y el programa paralelo basado en hilos pthread. Para ello se hace uso del Speed-up y la Ganancia.

$$S(n) = \frac{Tiempo\ ejecución\ secuencial(1)}{Tiempo\ ejecucion\ Pthread(n)}$$

La ganancia del sistema para un sistema con n procesadores se define como:

$$G(n) = \left(\frac{Tiempo\ ejecución\ secuencial(1)}{Tiempo\ ejecucion\ Pthread(n)} - 1\right) * 100$$

La eficiencia es una comparación del grado de Speed-up conseguido frente al valor máximo(valor ideal). Dado que 1 ≤ S(n) ≤ n, tenemos 1/n ≤ E(n) ≤ 1.

La eficiencia más baja (E(n) → 0 corresponde al caso en que todo el programa se ejecuta en un único procesador de forma serie. La eficiencia máxima (E(n) = 1) se obtiene cuando todos los procesadores están siendo completamente utilizados durante todo el periodo de ejecución. Se define como:

$$E(n) = \frac{Speed - UP(n)}{nº\ procesadores}$$

7.1. COMPARACIÓN CON PTHREAD

7.1.1. SPEED-UP

$$S(n) = \frac{Tiempo\ ejecución\ secuencial}{Tiempo\ ejecucion\ Pthread(n)}$$

Obtenidos los datos anteriores de tiempo de ejecución procedemos a mostrar los resultados:

$$Speed - Up(2\ hilos) = \frac{2.768531}{1.607356071} = 1{,}722413$$

$$Speed - Up(4\ hilos) = \frac{2.768531}{1.473448038} = 1{,}878947$$

$$Speed - Up(8\ hilos) = \frac{2.768531}{1.436833858} = 1{,}926827$$

$$Speed - Up(16\ hilos) = \frac{2.768531}{1.444459915} = 1{,}916655$$

7.1.2. GANANCIA

$$G(n) = \left(\frac{Tiempo\ ejecución\ secuencial(1)}{Tiempo\ ejecucion\ Pthread(n)} - 1\right) * 100$$

$$G(2\ Hilos) = (1{,}722413 - 1) * 100 = 72{,}24\ \%$$

$$G(4\ Hilos) = (1{,}878947 - 1) * 100 = 87{,}89\ \%$$

$$G(8\ Hilos) = (1{,}926827 - 1) * 100 = 92{,}27\ \%$$

$$G(16\ Hilos) = (1{,}916655 - 1) * 100 = 91{,}66\ \%$$

7.1.3. EFICIENCIA

$$E(n) = \frac{Speed - Up(n)}{nº\ procesadores} =$$

$$E(2) = \frac{1{,}722413}{2} = 0{,}8612$$

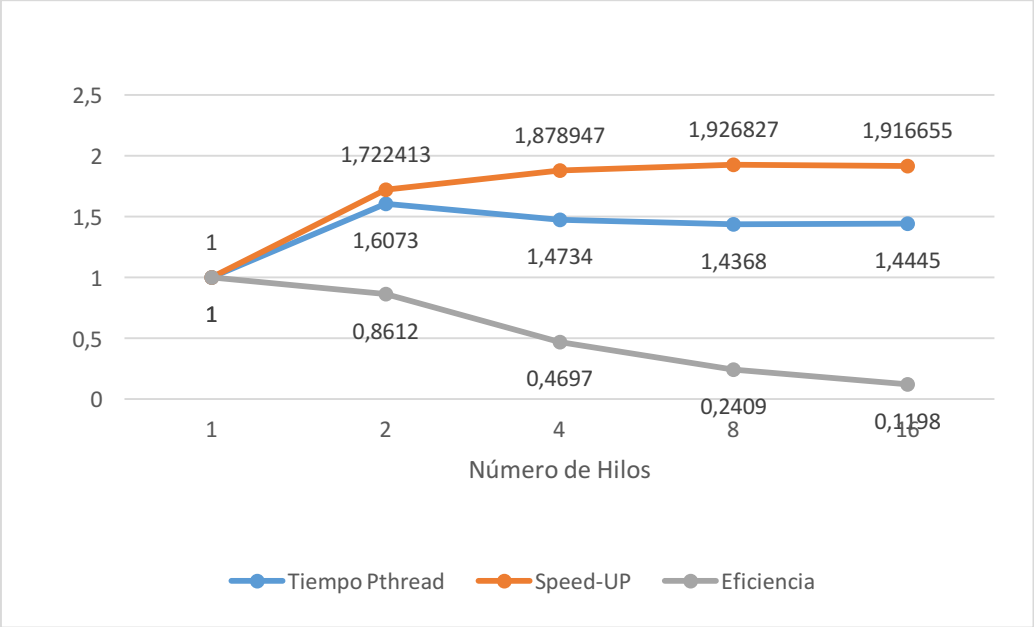
$$E(4) = \frac{1{,}878947}{4} = 0{,}4697$$

$$E(8) = \frac{1{,}926827}{8} = 0{,}2409$$

$$E(16) = \frac{1{,}916655}{16} = 0{,}1198$$

7.1.4. CONCLUSIÓN DE COMPARATIVA SECUENCIAL CON PTHREAD

En la gráfica mostrada a continuación se puede observar que en Speed-UP el punto más alto es con 8 hilos, esto quiere decir que este programa para esta computadora es el mejor rendimiento que se puede obtener.



Pero según la eficiencia respecto a los recursos de este ordenador indica que este programa es óptimo sólo para dos hilos y que a partir de ahí va bajando al usar más procesadores.

7.2. COMPARACIÓN CON OPENMP

7.2.1. SPEED-UP

$$S(n) = \frac{Tiempo\ ejecución\ secuencial}{Tiempo\ ejecucion\ Pthread(n)}$$

Obtenidos los datos anteriores de tiempo de ejecución procedemos a mostrar los resultados:

$$Speed - Up(2\ hilos) = \frac{2.768531}{1.730787039} = 1{,}5997$$

$$Speed - Up(4\ hilos) = \frac{2.768531}{1.611748934} = 1{,}7177$$

$$Speed - Up(8\ hilos) = \frac{2.768531}{1.547828913} = 1{,}7887$$

$$Speed - Up(16\ hilos) = \frac{2.768531}{1.573575974} = 1{,}7594$$

7.2.2. GANANCIA

$$G(n) = \left(\frac{Tiempo\ ejecución\ secuencial(1)}{Tiempo\ ejecucion\ Pthread(n)} - 1 \right) * 100$$

$$G(2\ Hilos) = (1{,}5997 - 1) * 100 = \ 59{,}97\ \%$$

$$G(4\ Hilos) = (1{,}7177 - 1) * 100 = 71{,}77\ \%$$

$$G(8\ Hilos) = (1{,}7887 - 1) * 100 = \ 78{,}87\ \%$$

$$G(16\ Hilos) = (1{,}7594 - 1) * 100 = 75{,}94\ \%$$

7.2.3. EFICIENCIA

$$E(n) = \frac{Speed - Up(n)}{nº\ procesadores} =$$

$$E(2) = \frac{1{,}5997}{2} = 0{,}8$$

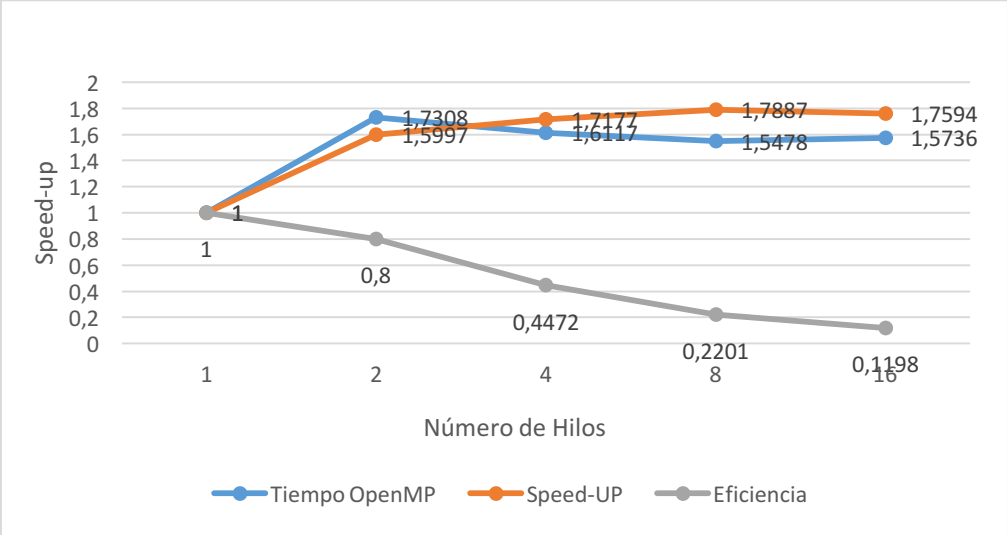
$$E(4) = \frac{1{,}7887}{4} = 0{,}4472$$

$$E(8) = \frac{1{,}7594}{8} = 0{,}22$$

$$E(16) = \frac{1{,}916655}{16} = 0{,}1198$$

7.2.4. CONCLUSIÓN DE COMPARATIVA SECUENCIAL CON OPENMP

En la gráfica mostrada a continuación se puede observar que en Speed-UP el punto más alto es con 8 hilos, eso quiere decir que este programa para esta computadora, el mejor rendimiento se obtiene con 8 hilos.



La eficiencia según la tabla muestra como con los recursos generados en el ordenador y el número de hilos usados, este programa es óptimo para 2 hilos y que a partir de ahí va bajando al usar más procesadores.

8. CONCLUSIÓN

Pthreads y OpenMP representan dos paradigmas de multiprocesamiento totalmente diferentes.

Desde un punto de vista personal este trabajo me ha parecido bastante bueno para conseguir varios objetivos como entender el trabajo de la memoria caché con paralelismo y memoria compartida entre hilos. Una vez comprobadas las dos formas de programar la paralelización he visto que OpenMP da muy buenos resultados y de forma más fácil, con solamente modificar o incluir unas directivas ya tienes el programa paralelizado, es bastante cómodo.

Desde el punto de vista técnico. Por un lado Pthreads es una API a un nivel muy bajo para trabajar con hilos. Por lo tanto, tiene un control muy preciso sobre la gestión de hilos (create/join), exclusiones mutuas, y así sucesivamente. Es bastante escueto.

Por otro lado, OpenMP es de nivel mucho más alto, más portátil y no lo limita a la utilización de C. También se programa mucho más fácil que pthreads. Un ejemplo específico de esto es el compartir las tareas que trae OpenMP, que le permiten dividir el trabajo a través de múltiples hilos con relativa facilidad.

9. BIBLIOGRAFÍA

http://mikiztli.blogspot.com.es/2008/03/mediciones-en-computo-paralelo.html

https://computing.llnl.gov/tutorials/pthreads/

https://es.wikipedia.org/wiki/OpenMP

http://openmp.org/wp/

https://en.wikipedia.org/wiki/POSIX_Threads

https://es.wikipedia.org/wiki/Caché_(informática)

http://www.fdi.ucm.es/profesor/jjruz/web2/temas/ec6.pdf

http://www.dia.eui.upm.es/Asignatu/arq_com/Paco/7-Cache.pdf

http://www.cpu-world.com/CPUs/Core_i5/Intel-Core%20i5%20Mobile%20i5-2430M.html