MASTER SIANI ULPGC

Computación Paralela

Paseo por las optimizaciones

Optimización basada en paralelismo multihilos

Sergio Marrero Marrero
Universidad de Las Palmas de Gran Canaria

14/03/2016

Índice

1. Optimización basada en paralelismo multihilos

1.1. Paralelizacion. Openmp

Como se recalcó en el ejercicio anterior, ahora es cuando tiene sentido hablar de paralelización, una vez que hemos conseguido que cada procesador funcione al máximo de su rendimiento. Para poder aplicar un paradigma de paralelización los elementos deben de ser independientes.

La siguiente imagen explica bastante bien en que consiste el paradigma:

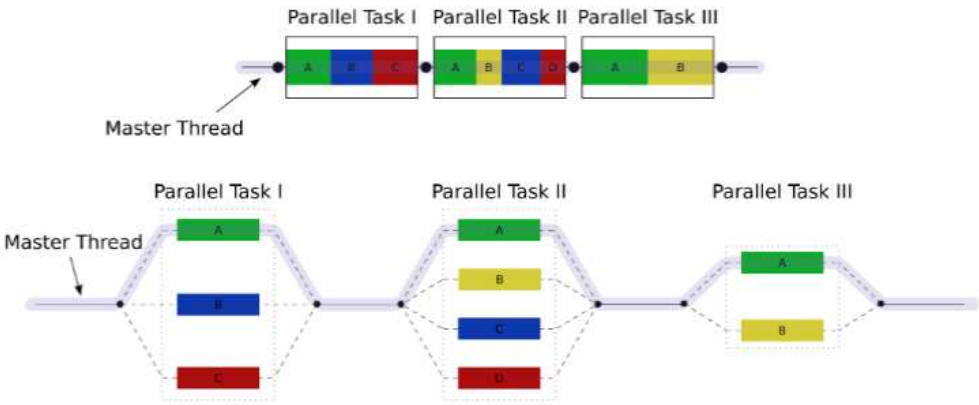

Figura 1: Paralelización en multihilos

En esta imagen se aprecia como existe un hilo principal, este hilo represeta a la ejecución secuencial delprograma. Llegados a un punto del código, se dividen las rutinas y se envían a distintos procesadores para que sean ejecutadas en paralelo. Cuando termina la etapa de paralelización se vuelve al hilo principal.

1.2. Compilación del código y análisis

El código que se deberá implementar es el siguiente:

```
void dgemm (int n, float* A, float* B, float* C){
```

```
int i,j,k;

#pragma omp parallel for

for ( j = 0; j < n; j += BLOCKSIZE )
        for ( i = 0; i < n; i += BLOCKSIZE )
                for ( k = 0; k < n; k += BLOCKSIZE)
                dgemm_blocking (n, i, j, k, A, B, C);
                }
```

Solo se ha añadido la parte del código que ha cambiado respecto la anterior. Lo único que se ha hecho es añadir la instrucción : # pragma omp parallel for, la cual le indica al compilador que el bucle for que viene a continuación puede procesarlo en distintos procesadores.

Para que esta funcione hay que incluir la librería:

```
#include <omp.h>
```

.

La forma de compilar esta función será la siguiente:

```
gcc -msse -fopenmp -o matrix6threads4096 matrix6threads4096.c -O3
```

La forma de ejecutar el archivo ejecutable es:

```
OMP_NUM_THREADS='numero de procesadores' ./matrix6threads4096
```

Los resultados de la ejecución del código son los siguientes:

Optimización	-gcc	tiempo(s)	MFlops/s	Archivo (Kb)	Speed up
Sin optimizar	-O	1284.4 (21.408)	107.0	8.7	—-
-O2	-O2	739.6 (12.327)	185.8	8.7	1
Par. datos	msse,-O3	174.9 (2.92)	785.7	8.7	4.2287
Segmentación	-O3	65.1 (1.09)	2111.5	8.7	11.361
Blocking	-O3	18	7627.7	8.7	41.089
openMP	msse,fopenmp-O3	6.6	20770	8.7	112.02

Cuadro 1: Optimizacion introduciondo openmp

Se observa que el speed-up supera al doble del anterior. Con todas las optimizaciones realizadas se observa que hemos mejorado el rendimiento del programa en cientos de veces respecto del mejor secuencial.

1.3. Speed-up respecto del algoritmo sin optimizar -O

Solo por curiosidad, calculemos rápidamente la mejora respecto del inicial.

$$\frac{tiemposecuencial}{fopenmp} = dfrac12846{,}6 = 194{,}55$$

Casi a un ritmo 200 veces superior.

1.4. Usar multihilos con distinto numero de procesadores

Se muestra en la siguiente tabla el tiempo de reloj empleado por distintas optimizaciones utilizando openmp (y el resto de optimizaciones) con distinto número de procesadores.

Procesadores	tiempo
1	17.9
2	11.2
3	8.1
4	6.6
5	6.4
6	6.3
7	5.8
8	5.3

Cuadro 2: Tiempos de reloj frente al número de procesadores

Se observa que si utilizamos un solo procesador, estamos ejecutando el programa 'Blocking'. Se comprueba que el tiempo es identico. Podemos además graficar esta tabla, y hacernos una idea de como evoluciona el rendimiento en función de los procesadores:

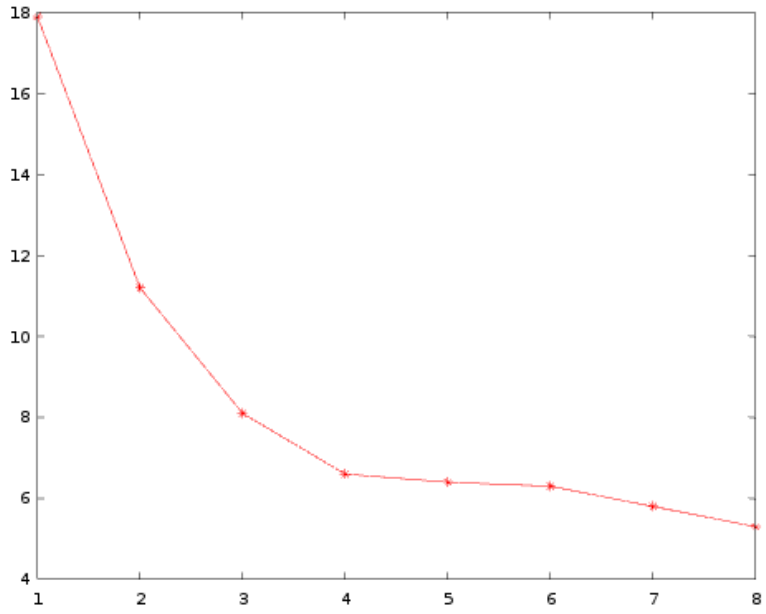

Figura 2: Tiempos de reloj frente al número de procesadores

Lo interesante de esta gráfica es observar que antes de usar 4 procesadores, la pendiente es casi constante y fuertemente decreciente. Sin embargo, a

partir de los 4 procesadores hay una fuerte variación y la pendiente deja de ser tan decreciente.Aunque sigue habiendo mejoria, no es tan significativa.Esto se debe a que realmente el ordenador tiene cuatro procesadores físicos. Cada procesador físico actúa como si fueran 2 puediendo ejecutar programas distintos a la vez. Esto acarrea un grave problema, y es que tienen que compartir los mismo niveles de memoria cache L1 y L2, por lo tanto la optimización deja de ser tan eficiente.

1.5. Evaluar el programa secuencial añadiendo multihilos y compararlo con el programa totalmente optimizado

A continuación se va utilizar el programa orgiginal y se modifica para que quede de la siguiente forma:

```
#include <omp.h>

...

void dgemm (int n, float* A, float* B, float* C)
{
 int i,j,k;
#pragma omp parallel for

 for (i = 0; i < n; ++i)
   for (j = 0; j < n; ++j)
   {
    float cij = C[i+j*n]; /* cij = C[i][j] */
    for(k = 0; k < n; k++ )
     cij += A[i+k*n] * B[k+j*n]; /* cij += A[i][k]*B[k][j] */
    C[i+j*n] = cij; /* C[i][j] = cij */
   }
}
```

Lo compilamos con la siguiente instrucción:

```
gcc -fopenmp -o matrix17open4096 matrix17open4096.c -O2
```

Y lo ejecutamos con esta otra

```
OMP_NUM_THREADS=4 ./matrix17open4096
```

y lo ejecutamos con 4 procesadores.

El resultado es de 154 s. Calculamos el speed-up respecto del programa completamente optimizado $speed - up = 154/6{,}6 = 23{,}3$ lo cual es bastante significativo. Los cuatro procesadores completamente optimizados tiene un rendimiento entorno a 23 veces superior frente al programa sin optimizaar corriendo en paralelo, ,lo cual justifica de sobra, la necesidad de optimizar previamente cada procesador.

1.6. Tabla característica del multiprocesador:

- Nombre del modelo comercial: *Core i7-3610 QM*
- Año en el que se empezó a comercializar: *2012 (abril)*
- Microarquitectura del multiprocesador: *Ivy Bridge*
- Microarquitectura del Core:*Ivy Bridge*
- Tecnología:22nm
- Número de cores:4
- Número de instrucciones que se envían en paralelo(Issue Width):8
- Tamaños de la memoria cache:L1=4x64KB;L2=4x256KB;L3=6MB