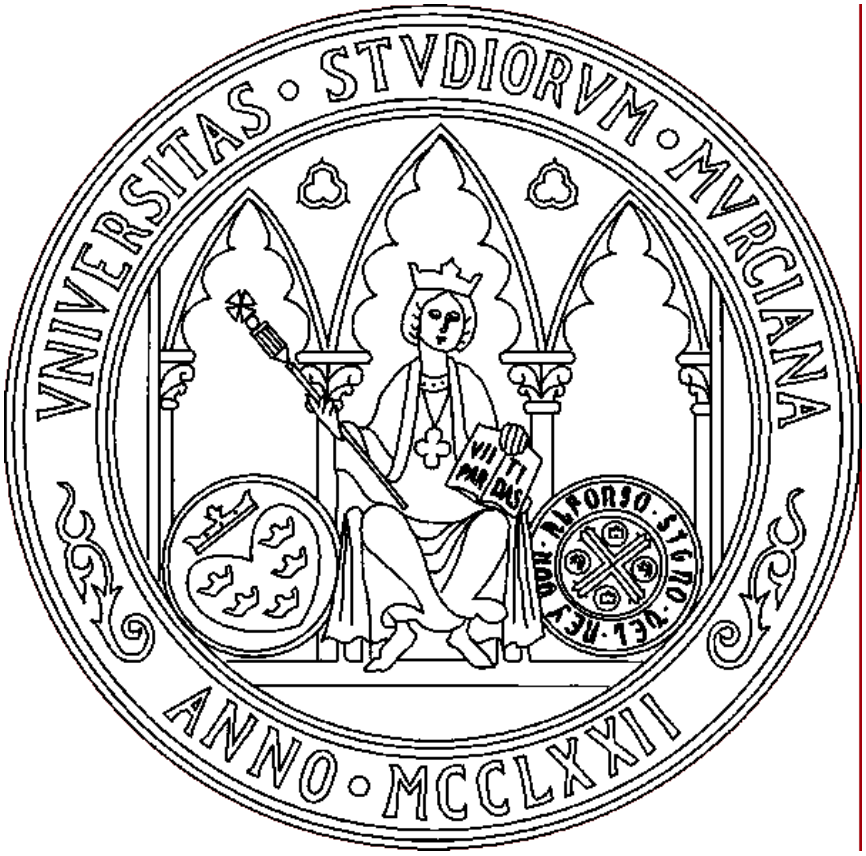
Programación de Arquitecturas Multinúcleo

Proyecto de Programación con OpenMP



Javier Barceló Pérez - Curso 2021/2022

Esta es la memoria del proyecto de programación de OpenMP de la asignatura de Programación de Arquitecturas Multinúcleo. En este directorio se incluyen varios archivos:

- SecMulMatCuaBlo.c → ejercicio 1
- SecMulMatBlo.c → ejercicio 2
- MulMatCuaBlo.c → ejercicio 3
- estudio.sh → ejecuta varias combinaciones diferentes de parámetros del ejercicio 3
- Makefile → compila todos los programas

Ejercicio 1

Se ha desarrollado el programa SecMulMatCuaBlo.c. En este programa se aprovecha en parte la funcionalidad de multiplicación de matrices usada en la entrega anterior de OpenMP, ya que apenas cambia esta funcionalidad. El programa crea tres matrices, las dos matrices factores m1 y m2, y la matriz resultado mr. Las tres son del mismo tamaño, nxn, siendo n un parámetro de entrada del programa. Se llama a una función mulmatcuablosec(), la cual recorre las submatrices de cada matriz, apuntando siempre al primer elemento de cada submatriz, y las multiplica usando la función mm() que aprovecha el código de la entrega anterior, cambiando únicamente el direccionamiento de las matrices. A esta función mm() se le envía como parámetro un puntero al primer elemento de una submatriz de cada una de las matrices, y multiplica estas submatrices de tamaño bxb. Se hace uso de una matriz auxiliar aux de tamaño bxb en la que se guarda el valor de la suma de las diferentes matrices que se multiplican con mm(), para luego insertar esa submatriz en la posición correspondiente en la función insertar_submatriz().

Ejercicio 2

Se ha usado como base el programa del ejercicio anterior, ya que para conseguir una multiplicación de matrices rectangulares solo hacía falta modificar algunos detalles de la implementación. El archivo es SecMulMatBlo.c. Se deben especificar las filas y columnas de las dos matrices de entrada, siendo la primera matriz de mxk y la segunda de kxn, para que la matriz resultado resulte de mxn. El algoritmo es el mismo, pero los accesos a las matrices cambian, y el manejo de punteros para calcular las submatrices se hace dependiendo del número de columnas de cada matriz, ya sea k o n si es m1 o m2, respectivamente.

Ejercicio 3

A partir del programa del ejercicio 1, se ha creado el programa MulMatCuaBlo.c, en el que se aprovecha las directrices de OpenMP para paralelizar el programa y conseguir mejores prestaciones. Para esto, se han paralelizado los bucles de la función mulmatcuablo y la función mm.

Esto se ha incluido en el bucle de mulmatcuablo():

```
#pragma omp parallel private(iam)

#pragma omp for private(i,j,suma) schedule(dynamic, 1)
```

Esto se ha incluido en el bucle de mm():

```
#pragma omp parallel for private(i,j,suma) schedule(dynamic, 1)
```

De esta forma, se paraleliza tanto las multiplicaciones de bloques como el recorrido en sí de los bloques, ya que cada multiplicación es independiente del resto.

Ejercicio 4

Para este ejercicio he probado diferentes conjuntos de tamaños de matriz, tamaños de bloque e hilos. He preparado un script que adjunto en el fichero comprimido llamado estudio.sh, en el que he ido modificando los conjuntos de valores que se recorren para la ejecución de la aplicación. La salida del script está guardada en el archivo output.txt con unos tamaños de matriz y bloque más pequeños, y en output2.txt con valores de tamaño de bloque y matriz más grandes. En los valores pequeños, la ejecución secuencial resultaba similar en eficacia que paralelizar. Sin embargo, con tamaños de problema más grande se conseguían más MFLOPS conforme aumentaban los hilos.

La conclusión a la que he llegado observando los resultados es que la relación entre estos tres parámetros afecta mucho al rendimiento. El tamaño del bloque acelera la ejecución para cualquier número de hilos. El tamaño de bloque, independientemente del tamaño del problema, siempre mejora la ejecución cuanto más grande es hasta cierto punto. Cuando se va acercando a la cuarta parte del tamaño del problema, b=n/4, el rendimiento deja de mejorar e incluso comienza a decaer. El aumento del número de hilos se hace notar mucho más cuanto más grande es el tamaño del problema.

A continuación una tabla con los valores del programa en MFLOPS cuando lo ejecuto para un tamaño de 1024, con tamaños de bloque de 2 a 256 y número de hilos de 1 a 12:

N=1024	T=1	T=2	T=3	T=6	T=9	T=11
B=2	0.017723	0.033108	0.051427	0.105443	0.136066	0.164175
B=4	0.135251	0.259102	0.397593	0.820030	1.080824	1.087895
B=8	0.731255	1.356385	2.323065	5.343273	5.558944	6.865979
B=16	2.379106	3.883513	7.421448	13.292537	13.367419	15.811374
B=32	2.248247	4.369315	7.281753	11.860900	11.942293	12.858063
B=64	1.636216	3.199324	4.391924	7.393711	6.846128	8.421370
B=128	1.452210	2.618354	3.555790	5.987430	5.491586	6.308647
B=256	1.422679	2.461471	3.416554	4.983891	5.005067	4.986774

Como aclaración, mi maquina tiene 6 cores con 2 núcleos por socket, por lo que las pruebas las he ido haciendo probando entre 1 y 12 hilos de ejecución.

Ejercicio 5

Apartado i)

Para este apartado, he dejado solo las directivas OpenMP de la función mm() que se encarga de paralelizar la multiplicación de dos bloques. Simplemente se consigue algo menos de rendimiento.

```
$ ./ejercicio5_1 1024 32 11

  Numero de hilos: 11
  Tam de matriz: 1024
  Tam de bloque: 32
    segundos: 0.187183, Mflops: 11.203779
```

Apartado ii)

Para este apartado, he quitado las directivas OpenMP de la función mm(), y he añadido al bucle de la j de la función mulmatcuablo() la siguiente directiva:

```
#pragma omp for private(i,j) schedule(dynamic,1)
```

Curiosamente, para tamaños de bloque y problema pequeños he obtenido resultados muy veloces, pero a la hora de escalar el problema a tamaños más grandes y sobre todo, tamaños de bloque más grandes, era absolutamente desastroso.

```
./ejercicio5_2 128 8 11

  Numero de hilos: 11
  Tam de matriz: 128
  Tam de bloque: 8
    segundos: 0.000487, Mflops: 67.228546

./ejercicio5_2 128 16 11

  Numero de hilos: 11
  Tam de matriz: 128
  Tam de bloque: 16
    segundos: 0.029986, Mflops: 1.092759
```

Apartado iii)

Para este apartado, partiendo del ejercicio 3, he quitado las directivas OpenMP de la función mm(). Los resultados son similares a lo esperado en el apartado i, se obtiene algo menos de rendimiento y la tendencia varía igual con el aumento del tamaño de bloque como con el ejercicio 3.

Ejercicio 6

Para este ejercicio se ha paralelizado el programa SecMulMatBlo.c de forma similar a la paralelización hecha en el ejercicio 3, creando un programa llamado MulMatBlo.c que paraleliza las filas de bloques y las multiplicaciones de bloques de matrices rectangulares, obteniendo mejores resultados que en su modelo secuencial.