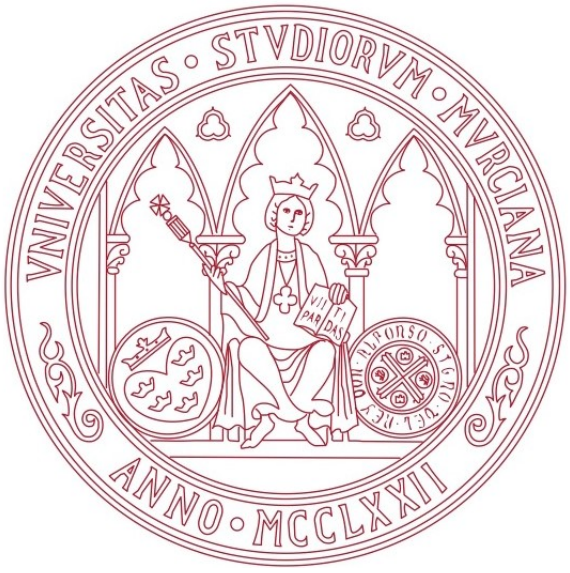


Programación de Arquitecturas Multinúcleo

PROYECTO DE PROGRAMACIÓN CON OPENMP

Francisco Arnaldo Boix Martínez - 52041159W

Marzo 2023

Índice

1. Programa `sec_mul_mat_cua_blo.c`

Este programa tiene como objetivo construir un algoritmo de multiplicación de matrices **cuadradas**, de forma **secuencial** y que el cómputo se haga en **bloques**. Para ello se supondrá que el tamaño de bloque es múltiplo del tamaño de las matrices y que todas las matrices tienen el mismo tamaño.

La idea detrás de este algoritmo reside en recorrer la matriz por los bloques (del tamaño especificado) y computar cada uno de esos bloques como una multiplicación de matrices convencional.

```
1  void multiply_matrix(double *a, int fa, int ca, int lda, double *b, int fb
       , int cb, int ldb, double *c, int fc, int cc, int ldc, int block_size)
2  {
3    int i, j, k, iam, nprocs;
4    double s;
5
6    assert(ca == fb);
7
8    int n = ca;
9    int num_blocks = n / block_size;
10
11   // recorro bloques y computo bloque
12   for (i = 0; i < num_blocks; i++)
13   {
14     for (j = 0; j < num_blocks; j++)
15     {
16       for (k = 0; k < num_blocks; k ++)
17         mult_submatrix(a, b, c, n, i, j, k, block_size);
18     }
19   }
20 }
```

A la hora de computar cada una de las submatrices, tenemos que llevar cuidado con los índices respecto a la matriz original.

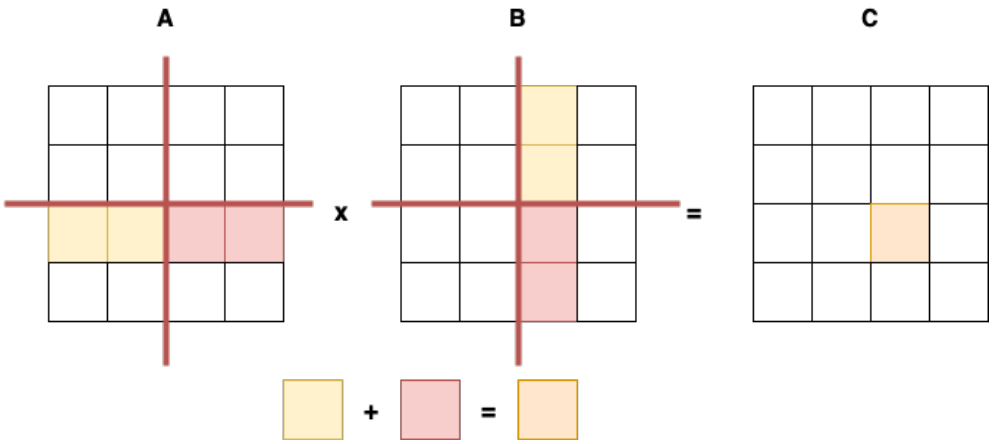


Figura 1: Cálculo de C_{ij} sumando resultados parciales

Para calcular C_{ij}, tenemos que calcular los resultados parciales de cada una de las submatrices e ir sumándolos. Si nos fijamos en la Figura 1, los colores amarillo y rojo de las matrices A y B respectivamente, representan que cuando se combinan ambos colores, generan el naranja.

```
void mult_submatrix(double *a, double *b, double *c, int n, int block_i,
    int block_j, int block_k, int block_size)
{
  int i, j, k;

  for (i = block_i * block_size; i < block_i * block_size + block_size; i
   ++)
  {
    for (j = block_j * block_size; j < block_j * block_size + block_size;
   j++)
    {
      for (k = block_k * block_size; k < block_k * block_size + block_size
   ; k++)
        c[i * n + j] += a[i * n + k] * b[k * n + j];
    }
  }
}
```

Es importante recalcar, que el valor de C_{ij} dentro del cómputo de la submatriz **no** se inicializa a 0 puesto que a la hora de reservar el espacio de memoria, he usado `calloc`. El resto de matrices, se inicializan posteriormente de forma **pseudo-aleatoria**.

```
  a = (double *)malloc(sizeof(double) * t * t);
  b = (double *)malloc(sizeof(double) * t * t);
  c = (double *)calloc(t * t, sizeof(double));
```

2. Programa `sec_mul_mat_blo.c`

Si nos fijamos en el algoritmo de la sección pasada, podemos observar que requería de un desglose de los tamaños bastante superior al que se precisaba en aquel momento. Esto quiere decir que desde el primer momento se tenía en cuenta que las matrices podían no ser cuadradas, y eso es precisamente lo que se busca conseguir con este programa.

Para poder hacer totalmente funcional el algoritmo de multiplicación de matrices por bloques para matrices que no tienen porqué ser cuadradas, tenemos que hacer un par de ajustes, pero la idea principal sigue siendo la misma. Es importante asegurarse que las **columnas de A** y las **filas de B** tengan el mismo tamaño, de lo contrario no se pueden multiplicar. Ver Figura 2.

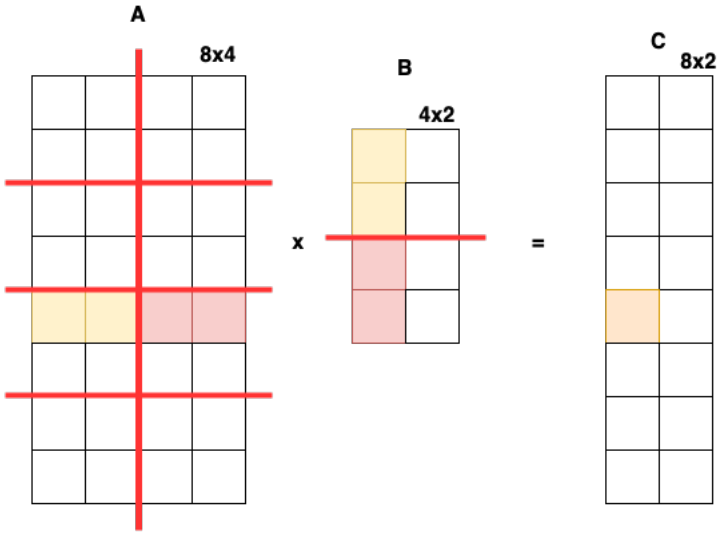


Figura 2: Cálculo de C_{ij} en matrices no cuadradas

```
void multiply_matrix(double *a, int fa, int ca, int lda, double *b, int fb
    , int cb, int ldb, double *c, int fc, int cc, int ldc, int block_size)
{

  int i, j, k, iam, nprocs;
  double s;

  assert(ca == fb);      // Aseguramos se pueden multiplicar

  int num_blocks_fa = fa / block_size;
  int num_blocks_cb = cb / block_size;
  int num_blocks_ca = ca / block_size;

  // recorro bloques y computo bloque
  for (i = 0; i < num_blocks_fa; i++)
  {
    for (j = 0; j < num_blocks_cb; j++)
    {
      for (k = 0; k < num_blocks_ca; k++)
        mult_submatrix(a, b, c, i, j, k, block_size, fa, cb, ca);
    }
  }
}
```

La diferencia fundamental respecto a la versión anterior, es que ahora el tamaño no es el mismo en ambas matrices. Por tanto, el número de bloques no será el mismo para todas las matrices.

Del mismo modo, a la hora de computar cada uno de los bloques, pese a que sean bloques cuadrados, tenemos que tener en cuenta la diferencia de tamaños entre las matrices para poder acceder a ellas a la hora de ir calculando los resultados parciales.

```
void mult_submatrix(double *a, double *b, double *c, int block_i, int
    block_j, int block_k, int block_size, int m1, int n2, int n1)
{
  int i, j, k;


  for (i = block_i * block_size; i < block_i * block_size + block_size; i
   ++)
  {
    for (j = block_j * block_size; j < block_j * block_size + block_size;
   j++)
    {
      for (k = block_k * block_size; k < block_k * block_size + block_size
   ; k++)
        c[i * n2 + j] += a[i * n1 + k] * b[k * n2 + j];
    }
  }
}
```

También he tenido que hacer algunos ajustes con los tamaños en algunas funciones auxiliares que tenía dentro del programa, debido a que estas asumían que las matrices eran cuadradas. Un ejemplo podría ser la función de testing, la cual antes estaba declarada del siguiente modo:

```
int test(double *m, double **global, int n);
```

Ahora pasa a tener la siguiente declaración:

```
int test(double *m, double **global, int t1, int t2);
```

3. Programa `mul_mat_cua_blo.c`

La sintaxis de uso del programa es la siguiente:

`./mul_mat_cua_blo.c <dim_mat_n> <tam_blo_b> <num_threads>`

Partiendo de la versión de la sección 1, es decir, asumiendo que las matrices son **cuadradas**, haremos una versión **paralela** usando `OpenMP`. El reparto se hará a nivel de bloque. Es decir, cada hilo computará un bloque.

Para ello, simplemente tendremos que añadir las cláusulas omp pertinentes. Veámoslas:

```
void multiply_matrix(double *a, int fa, int ca, int lda, double *b, int fb
    , int cb, int ldb, double *c, int fc, int cc, int ldc, int block_size)
{
...
#pragma omp parallel private(iam, i, j, k)
  {
#if defined(_OPENMP)
    iam = omp_get_thread_num();
#endif

#pragma omp for collapse(2)
    for (i = 0; i < num_blocks; i++)
    {
      for (j = 0; j < num_blocks; j++)
      {
        for (k = 0; k < num_blocks; k++)
          mult_submatrix(a, b, c, n, i, j, k, block_size);
      }
    }
  }
}
```

Creamos una región paralela, y dentro de la misma definimos un `pragma omp for`, con el scheduler por defecto. El reparto, como podemos ver se hace a nivel de bloque puesto que es lo que computa la función `mult_submatrix`. Cada uno de los hilos ejecutará esa función. Si hay más bloques a computar que hilos disponibles, cada hilo computará mas de un bloque.

Previo a la llamada a esta función he tenido que definir el número de hilos de `OpenMP`, para que ignore la variable de entorno `$OMP_NUM_THREADS`. Para ello, bastará con agregar lo siguiente al programa:

```
omp_set_num_threads(threads);
```

Para comprobar que el reparto se está haciendo correctamente, véase la Sección6.

4. Programa `mul_mat_blo.c`

Del mismo modo que hemos hecho en la Sección 3, tenemos que partir de la versión en secuencial de la multiplicación de matrices no cuadradas, y agregarle paralelismo. Para ello, he añadido los mismos pragmas que en ese caso.

Para comprobar que el reparto se está haciendo correctamente, véase la siguiente Sección (6).

5. Testing

Para cada uno de los programas descritos a lo largo de este documento, se puede definir la constante `TEST`. De este modo, se compara el resultado del algoritmo en cuestión frente a una multiplicación de matrices convencional de forma secuencial.

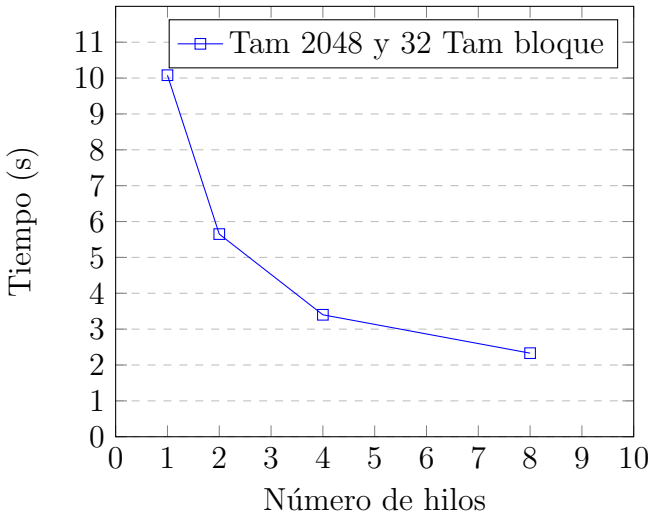
```
// Inicio del programa
// Includes

#define TEST
...
```

6. Estudio experimental de `mul_mat_cua_blo.c`

Cabe resaltar que he hecho un ligero cambio en el número de hilos para el estudio, para ajustarlo lo máximo posible a la máquina donde se han ejecutado los casos de prueba. En concreto, mi máquina (Apple Macbook Air M1 2020), cuenta con 8 núcleos físicos sin Hyperthreading, por tanto no tiene sentido ejecutar con más de 8 hilos para las pruebas. De hecho me parecía aún mas interesante compararlo con una versión en secuencial. Véase la Tabla 1.

Pasemos a ver las conclusiones para cada uno de los distintos parámetros comparados:

- **Nº Hilos:** Podemos observar que conforme vamos aumentando el número de hilos, el tiempo va disminuyendo en un factor de n, siendo n el número de hilos. Evidentemente, como sabemos por los problemas de la **programación concurrente** y/o la **Ley de Amdhal**, no siempre conseguiremos esa reducción de un factor de n, pero aún así vemos la tendencia de la curva.



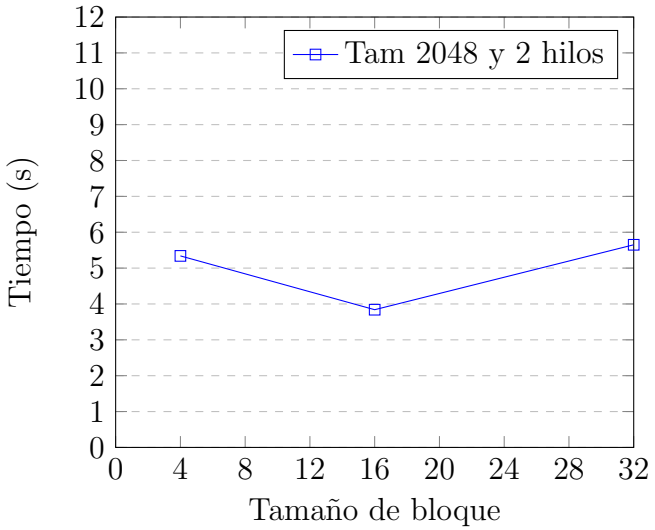
- **Tamaño de bloque:** Como podemos comprobar, para un mismo número tamaño de matriz, con el mismo número de hilos, obtenemos los resultados mostrados en la Figura 6.

  Como podemos observar, si aumentamos el tamaño de bloque de 4 a 16 obtenemos una mejora de rendimiento, sin embargo si volvemos a doblar su tamaño, observamos una reducción del mismo. Incluso observamos que con un tamaño de bloque de 32 obtenemos **peores** resultados que con 4. Esto tiene mucho que ver con la forma y el diseño de la caché del procesador **M1 ARM**.

  No he sido capaz de encontrar información suficientemente detallada acerca de esta caché, pero lo mas probable es que las líneas de caché tengan un tamaño tal, que funcionan mejor para un tamaño de bloque de 16 elementos. Esto permite explotar la **localidad espacial** y **temporal**.

  Lo que es posible que esté pasando es que con un tamaño de 4x4, no llenemos suficientemente la caché para computar un elemento C_{ij} de la matriz resultado, y sin embargo, con un tamaño de 32, se nos agote el tamaño de la caché y obtengamos algún fallo de caché por falta de espacio.

  Por otro lado, con un tamaño de 16, es posible que sea idóneo **para esta máquina en concreto**.



Threads	Size	Block Size	Time
1	512	4	0.09
		16	0.05
		32	0.09
	1024	4	0.60
		16	0.85
		32	1.43
	2048	4	10.47
		16	6.89
		32	10.08
2	512	4	0.03
		16	0.03
		32	0.05
	1024	4	0.31
		16	0.44
		32	0.76
	2048	4	5.34
		16	3.84
		32	5.65
4	512	4	0.02
		16	0.01
		32	0.02
	1024	4	0.16
		16	0.29
		32	0.53
	2048	4	2.81
		16	2.50
		32	3.40
8	512	4	0.02
		16	0.01
		32	0.02
	1024	4	0.14
		16	0.16
		32	0.28
	2048	4	2.10
		16	1.55
		32	2.33

Cuadro 1: Estudio experimental