PROGRAMACIÓN PARALELA Y COMPUTACIÓN DE ALTAS PRESTACIONES

TRABAJO AUTÓNOMO: ENTORNOS DE PROGRAMACIÓN PARALELA

PRÁCTICAS DE PPCAP 17/18

José Luis Cánovas Sánchez

LIBRERÍAS (MKL)

CUESTIÓN 1

Comparar prestaciones de multiplicación secuencial por bloques y sin bloques con la multiplicación de MKL.

En el directorio `1/` adjunto, o en `mpp17-45/mkl/matriz_matriz/` de luna, se encuentra el código fuente de las versiones con y sin bloques, y MKL. El código de las 2 primeras es el usado en las otras prácticas, siendo el sin bloques la versión con transpuesta que dio los mejores tiempos. El código de MKL utiliza las llamadas a BLAS de primer, segundo y tercer nivel:

De nivel 1: producto escalar

```
1.  double *bt = (double *) malloc(sizeof(double)*cb*fb);
2.  trasponer_matriz_esp(b,fb,cb,ldb,bt,cb,fb,fb);
3.
4.  for(i=0;i<fa;i++)
5.  {
6.    for(j=0;j<cb;j++)
7.    {
8.      c[i*ldc+j] = cblas_ddot (ca, &a[i*lda], 1, &bt[j*fb], 1);
9.    }
10. }
```

Recorremos las filas de a y columnas de b y realizamos los productos escalares con BLAS.

De nivel 2: matriz por vector

```
1.  double *bt = (double *) malloc(sizeof(double)*cb*fb);
2.  trasponer_matriz_esp(b,fb,cb,ldb,bt,cb,fb,fb);
3.  // y := alpha*A*x + beta*y   => y := A*x
4.  // fila_i_c = fila_i_a * b = b' * fila_i_a
5.  // CblasTrans => nos transpone b  // CblasNoTrans => ya se lo damos transpuesto
6.  for(i=0;i<fa;i++)
7.  {
8.    cblas_dgemv (CblasRowMajor, CblasNoTrans, cb, fb, alpha, b, fb, &a[i*lda], 1, beta, &c[i*l
    dc], 1);
9.  }
```

Recorremos las filas de a, y usando el producto de matriz por vector de BLAS, con la propiedad de que la traspuesta del producto es el producto de las traspuestas con orden inverso, de modo que transponiendo b, podemos realizar todas las operaciones aprovechando la localidad espacial de cada variable.

De nivel 3: matriz por matriz

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, fa, cb, ca, alpha, a, lda, b, ldb, beta,
c, ldc);
```

Donde se indica que se ordena la matriz en memoria por filas, que no es la transpuesta, se pasan las dimensiones, y las constantes alpha y beta que indican en este caso, alpha = 1 y beta = 0.

$$C \Leftarrow \alpha A * B + \beta C$$

En una primera prueba utilicé gcc para las versiones sin MKL, y no indiqué explícitamente que se usara un único hilo en MKL. Por referencia, los tiempos tomados en Saturno de esa primera ejecución son:

N =	2000	3000	4000
Sin bloques (trans)	22.665726	75.494501	177.077932
Con bloques			
TB=25	13.496091	42.318879	100.471517
TB=50	*12.777352*	*40.772234*	*95.693469*
TB=100	14.281152	45.684251	112.802174
MKL Level 3	**0.878072**	**1.149646**	**2.310619**

Vemos que en la versión de bloques, con tamaño de bloque 50 se obtiene el mejor tiempo, pero frente a la implementación de MKL de la multiplicación de matrices BLAS, ésta última llega a ser ~40 veces más rápida que la versión por bloques.

Una vez recompilo con icc todas las versiones, y pongo explícitamente las líneas

```
1.  mkl_set_dynamic( 0 );
2.  mkl_set_num_threads( 1 );
```

para forzar el uso de un solo hilo en MKL, tomo los tiempos en Júpiter, pues Saturno estaba ocupado por la cola de investigación.

N =	2000	3000	4000
Sin bloques (trans)	6.292005	22.071910	52.246046
Con bloques			
TB=25	8.572358	28.787944	70.131146
TB=50	6.983950	23.548761	56.315092
TB=100	6.790730	22.259663	52.701967
MKL Level 1	6.363639	21.103719	50.048953
MKL Level 2	5.457693	17.092290	40.428506
MKL Level 3	**1.293702**	**3.669136**	**7.389529**

En este caso la diferencia de MKL Level 3 con el mejor tiempo secuencial es *solo* hasta ~7 veces más rápido que las otras implementaciones secuenciales, no las ~40 veces de antes.

Comparando las versiones con versiones inferiores de BLAS, vemos que la versión Level 1 (producto escalar), supera por poco a nuestra secuencial con transpuesta, aunque con tamaños pequeños, el overhead de llamar a BLAS parece que nos da ventaja, pero no mucha.

Utilizando un Level 2 de BLAS ya obtenemos tiempos significativamente mejores, pero aún lejos de las velocidades que se obtienen con el Level 3 implementado por MKL.

CUESTIÓN 2

En la rutina de factorización LU por bloques, sustituir las multiplicaciones de matrices por llamadas a BLAS. Comparar tiempos de ejecución. Comparar los resultados con los de la factorización usando directamente MKL.

En el directorio `2/` adjunto, o en `mpp17-45/mkl/lu/` de luna, se encuentra el código fuente de las versiones de LU por bloques con llamadas a BLAS de MKL en la multiplicación de matrices, y la LU de MKL.

En la primera versión, sustituimos la multiplicación con resta de matrices por el siguiente código, donde la alfa y beta corresponden a -1 y 1 respectivamente, para restar el producto sobre la matriz c:

$$C \Leftarrow \alpha A * B + \beta C$$

```
1.  void multiplicar_restar_matrices(double *a,int fa,int ca,int lda,double *b,int fb,int cb,int
     ldb,double *c,int fc,int cc,int ldc)
2.  {
3.    mkl_set_dynamic( 0 );
4.    mkl_set_num_threads( 1 );
5.    double alpha = -1.0;
6.    double beta = 1.0;
7.    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, fa, cb, ca, alpha, a, lda, b, ldb,
    beta, c, ldc);
8.  }
```

En la versión de la LU de LAPACK en MKL, sólo hace falta llamar a la siguiente función, en vez de a nuestra función de LU por bloques. La variable ipiv es el array de cambios de filas, pues en verdad se hace la descomposición PA = LU, donde se cambian las filas de A para minimizar los errores al operar con valores pequeños.

```
LAPACKE_dgetrf(LAPACK_ROW_MAJOR, fa, ca, a, lda, ipiv);
```

Comparamos con la tabla de tiempos (segundos) del código secuencial por bloques de referencia, la versión con MKL para el producto de matrices, y la versión de MKL. Usamos bloques de tamaño 10, pues en pruebas anteriores son los que mejor tiempo obtenían. Todos compilados con icc. Ejecutado en Júpiter:

N=	5000	7000	9000
LU BLOQUES 10	61.293587	168.515420	359.437214
LU BLOQUES 10 + MKL	6.558937	17.514004	36.840648
LU MKL	**5.908207**	**12.960004**	**27.409984**

Vemos cómo el mayor tiempo de cómputo en la LU por bloques es la multiplicación de matrices. En la versión LU en que se utiliza MKL para las multiplicaciones, vemos una mejora de ~10 veces en el tiempo respecto a la versión de referencia. Esto concuerda con la gran mejora que obteníamos en el ejercicio anterior al usar MKL. Y al usar la versión de MKL de la LU de LAPACK mejoramos a la LU por bloques en varios segundos.

CUESTIÓN 3

Realizar dos de los siguientes: experimentos con LU con paralelismo OpenMP+MKL, con LU con paralelismo MPI+MKL, LU combinando MKL en CPU con GPU, LU combinando MKL en CPU con Xeon Phi, experimentos de LU con PARDISO.

En el directorio `3/` adjunto, o en `mpp17-45/mkl/lu_mkl_openmp/` de luna, se encuentra el código fuente de las paralelizaciones.

Las versiones programadas son:

- LU bloques + MKL + OpenMP (Marte):

En la función de producto con resta de matrices lanzamos 6 hilos con OpenMP, los cuales dividen, basado en su número de hilo, la matriz A en bloques de filas y con BLAS de MKL realizan el producto de cada pedazo de A por B.

```
1.  void multiplicar_restar_matrices(double *a,int fa,int ca,int lda,double *b,int fb,int cb,int
     ldb,double *c,int fc,int cc,int ldc)
2.  {
3.    double alpha = -1.0;
4.    double beta = 1.0;
5.
6.    #pragma omp parallel
7.    {
8.      int t = omp_get_thread_num();
9.      int numt = omp_get_num_threads();
10.
11.     double * da = &a[ t * lda * (fa / numt) ];
12.     double * dc = &c[ t * ldc * (fc / numt) ];
13.     int dfa = (t != numt-1) ? (fa / numt) : (fa / numt) + fa%numt;
14.
15.     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dfa, cb, ca, alpha, da, lda, b, l
    db, beta, dc, ldc);
16.   }
17.
18. }
```

También se paralelizan con un parallel for de OpenMP el bucle más externo de la resolución de los sistemas de ecuaciones en la LU.

Los tiempos, en segundos, medidos en Marte para distintos tamaños de bloque y matrices se muestran abajo:

N=	7000	9000	11000
Bloque 10	14.670943	31.056077	55.859629
Bloque 50	4.711536	9.719458	17.473662
Bloque 100	**4.381563**	**8.967805**	**16.121736**
Bloque 150	4.565176	9.211762	16.331938
Bloque 200	4.580018	9.279753	16.334551
Bloque 250	4.804477	9.558276	16.809873

Vemos que los mejores tiempos se obtienen con bloques de tamaño 100, seguido de cerca tamaños algo mayores. Con bloques de tamaño 10 los tiempos son ~3.5 veces más lentos, mostrando una elección de tamaño de bloque inadecuado para la técnica y máquina.

Con un código de comprobación de la LU cedido por Adrián, comprobamos que todas las ejecuciones son correctas utilizando la implementación directa de MKL y deshaciendo los cambios de filas de P. Todas las pruebas tenían un error inferior a 10^-7.

- LU MKL multihilo (Saturno):

Al intentar reservar 48 hilos en Saturno el sistema de colas me lanzaba un error, así que solo he podido reservar 24 hilos. Esta prueba sirve como comparación para la descomposición LU de MKL en Xeon Phi que viene a continuación.

Los tiempos en Saturno son (segundos):

7000	9000	11000	13000	15000
2.283228	5.050476	7.746598	13.870530	18.148106

- LU MKL Offload (Venus + Xeon Phi):

En esta versión llamamos a la misma función LAPACK de MKL de la versión anterior pero dentro de una sección que se envía al Xeon Phi con el pragma:

```
1.  #pragma offload target(mic) inout(a:length(fa*ca)) inout(ipiv:length(fa))
```

Medimos los tiempos en segundos en Venus:

7000	9000	11000	13000	15000
6.291781	7.888072	9.463816	12.587179	20.139830

Comparado con la versión ejecutada en Saturno, vemos cómo a menor tamaño de matriz, el coste de ejecutarlo en Xeon Phi es mayor que en el Intel Xeon de Saturno, donde no hay coste de copiar las matrices a y desde el Xeon Phi, y donde la ejecución secuencial que tenga el algoritmo de la LU utilizado se beneficiará de los cores del Intel Xeon. Cuando aumentamos el tamaño de la matriz, vemos cómo la diferencia en tiempos se reduce significativamente, y podemos suponer que con matrices suficientemente grandes, el overhead del Xeon Phi será menor que el beneficio obtenido por el mayor número de cores.