```
#if !defined(TIMING_H_)
#define TIMING_H_ 1

#include <time.h>

#if defined(__cplusplus)
extern "C" {
#endif

long double timespec_to_dbl (struct timespec x);
long double timespec_diff (struct timespec start, struct timespec finish);

long double timer_resolution (void);
void get_time (struct timespec*);

#if defined(__cplusplus)
}
#endif

#endif // TIMING_H_
```



```
/*
  Idealmente, no harÃ-a falta hacer cambios en este archivo. Puede querer
  hacer algunos cambios para depurar el programa, pero recuerde revertirlos
  para las corridas finales.

  El formato de salida: "Tam: %u\tmflop/s: %g\n"

El archivo ha pasado por estas manos:
    Jason Riedy
    David Bindel
    David Garmire
    Juan Heguiabehere
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <float.h>
#include <math.h>

#include <sys/types.h>
#include <sys/resource.h>

#include <unistd.h>
#include <time.h>

#include "timing.h"

#if !defined(COMPILER)
#  define COMPILER "desconocido"
#endif
#if !defined(FLAGS)
#  define FLAGS "desconocido"
#endif
#if !defined(PLATFORM)
#  define PLATFORM "desconocido"
#endif

/*
  La firma de su funciÃ³n DEBE ser la siguiente:
*/
extern const char* matmult_desc;
extern void matmult_cuadrado ();

/*
  Tratamos de hacer suficiente cantidad de iteraciones para conseguir
  mediciones razonables. Las matrices se multiplican al menos MIN_TIMES veces.
  Si eso no tarda al menos MIN_SECS segundos, duplicamos el nÃºmero de iteracion
es
  y probamos de nuevo.

  Puede jugar con estos parÃ¡metros para apurar el debugging...
*/
#define MIN_RUNS 4
#define MIN_SECS 1.0

/*
  La lista de tamaÃ±os de matriz es un poco exÃ³tica... hay efectos interesantes
  cerca de algunas potencias de 2.
*/
const int test_sizes[] = {
```

```
  31, 32, 96, 97, 127, 128, 129, 191, 192, 229,
#if defined(DEBUG_RUN)
# define MAX_SIZE 229u
#else
  255, 256, 257, 319, 320, 321, 417, 479, 480, 511, 512, 639, 640,
  767, 768, 769,
# define MAX_SIZE 769u
#endif
};

#define N_SIZES (sizeof (test_sizes) / sizeof (int))

static double A[MAX_SIZE * MAX_SIZE], B[MAX_SIZE * MAX_SIZE],
  C[MAX_SIZE * MAX_SIZE];

static void matriz_init (double* A);
static void matriz_clear (double* C);

/* Compara C con la version de multiplicar A*B con los tres bucles*/
static void validar_matmult (const int M,
                             const double *A, const double *B, double *C);
/* Contar los mflop/s */
static double cronometrar_matmult (const int M,
                           const double *A, const double *B, double *C);

int
main (void)
{
  int i;
  double mflop_s;

  matriz_init (A);
  matriz_init (B);

  printf ("Compilador:\t%s\nOpciones:\t%s\nPlataforma:\t%s\nDescripcion:\t%s\n\n",
          COMPILER, FLAGS, PLATFORM, matmult_desc);

  /* printf ("Resolucion: %Lg\n", timer_resolution() ); */

  for (i = 0; i < N_SIZES; ++i) {

    const int M = test_sizes[i];

#ifndef NO_VALIDATE
    validar_matmult (M, A, B, C);
#endif
    mflop_s = cronometrar_matmult(M, A, B, C);

    printf ("Tam: %u\tmflop/s: %lg\n", M, mflop_s);
  }

  return 0;
}

void
matriz_init (double *A)
{
  int i;

  for (i = 0; i < MAX_SIZE*MAX_SIZE; ++i) {
    A[i] = ((unsigned char)lrand48()) / 20 ; /* drand48 (); */
  }
```



```
}

void
matriz_clear (double *C)
{
  memset (C, 0, MAX_SIZE * MAX_SIZE * sizeof (double));
}

/*
  Los productos de matrices satisfacen la siguiente cota de error:
  float(sum a_i * b_i) = sum a_i * b_i * (1 + delta_i)
  donde delta_i <= n * epsilon. Para validar su producto de matrices,
  computamos cada elemento y verificamos que su producto estÃ© dentro
  de un rango de error de tres veces este valor. Lo hacemos tres veces
  porque hay tres fuentes de error:

  - El error de redondeo en su producto
  - El error de redondeo en nuestro producto
  - El error de redondeo al computar la cota de error (esta Ãºltima no es tan si
gnificativa)
*/
void
validar_matmult (const int M,
                 const double *A, const double *B, double *C)
{
  int i, j, k;

  matriz_clear (C);
  matmult_cuadrado (M, A, B, C);

  for (i = 0; i < M; ++i) {
    for (j = 0; j < M; ++j) {

      double dotprod = 0;
      double errorbound = 0;
      double err;

      for (k = 0; k < M; ++k) {
        double prod = A[k*M + i] * B[j*M + k];
        dotprod += prod;
        errorbound += fabs(prod);
      }
      errorbound *= (M * DBL_EPSILON);

      err = fabs(C[j*M + i] - dotprod);
      if (err > 3*errorbound) {
        printf("Error en el producto de matrices.\n");
        printf("C(%d,%d) deberia ser %lg, y fue %lg\n", i, j,
               dotprod, C[j*M + i]);
        printf("Error de %lg, limite aceptable %lg\n",
               err, 3*errorbound);
        exit(-1);
      }
    }
  }
}

double
cronometrar_matmult (const int M,
             const double *A, const double *B, double *C)
{
  struct timespec start, finish;
```

```
  double mflops, mflop_s;
  double secs = -1.0;

  int num_iterations = MIN_RUNS;
  int i;

  while (secs < MIN_SECS) {

    matriz_clear (C);
    get_time (&start);
    for (i = 0; i < num_iterations; ++i) {
      matmult_cuadrado (M, A, B, C);
    }
    get_time (&finish);
    secs = timespec_diff (start, finish);

    mflops  = 2.0 * num_iterations * M * M * M / 1.0e6;
    mflop_s = mflops/secs;

    num_iterations *= 2;
  }

  return mflop_s;
}
```



```
const char* matmult_desc = "Multiplicacion por 3 bucles invertido.";

void
matmult_cuadrado (const int M,
                  const double *A, const double *B, double *C)
{
  int i, j, k;

  for (j = 0; j < M; ++j) {
       for (i = 0; i < M; ++i) {
            const double *Ai_ = A + i;
            const double *B_j = B + j*M;

            double cij = *(C + j*M + i);

            for (k = 0; k < M; ++k) {
                 cij += *(Ai_ + k*M) * *(B_j + k);
            }

            *(C + j*M + i) = cij;
       }
  }
}
```

```
const char* matmult_desc = "Multiplicacion basica de tres bucles.";

void
matmult_cuadrado (const int M,
             const double *A, const double *B, double *C)
{
  int i, j, k;

  for (i = 0; i < M; ++i) {
       for (j = 0; j < M; ++j) {
            const double *Ai_ = A + i;
            const double *B_j = B + j*M;

            double cij = *(C + j*M + i);

            for (k = 0; k < M; ++k) {
                 cij += *(Ai_ + k*M) * *(B_j + k);
            }

            *(C + j*M + i) = cij;
       }
  }
}
```



```
#include <stdlib.h>

const char* matmult_desc = "Multiplicacion basica de tres bucles luego de trasponer.";

double* trasp_init( const int N, const double* M )
{
        int i,j;
        double* buf = malloc( sizeof(double)*N*N );
        for( i = 0; i < N; i++ )
                for( j = 0; j < N;  j++ )
                        buf[i*N + j] = M[j*N + i];
        return buf;
}

void trasp_free( double* M )
{
        free( M );
}

void
matmult_cuadrado (const int M,
             const double *A, const double *B, double *C)
{
  int i, j, k;

  double* T = trasp_init( M, A );
  for (i = 0; i < M; ++i) {
       for (j = 0; j < M; ++j) {
            const double *Ai_ = T + i*M;
            const double *B_j = B + j*M;

            double cij = *(C + j*M + i);

            for (k = 0; k < M; ++k) {
                 cij += *(Ai_ + k) * *(B_j + k);
            }

            *(C + j*M + i) = cij;
       }
  }
  trasp_free( T );
}
```

```
const char* matmult_desc = "Multiplicacion de matrices simple por bloques.";

/* Se puede jugar con este valor */
#if !defined(TAM_BLOQUE)
#define TAM_BLOQUE ((int) 56)
#endif

/*
  A es M x K
  B es K x N
  C es M x N

  lda es el "lado" de la matriz original (el M de la matriz cuadrada).
*/

void
matmult_basico (const int lda,
             const int M, const int N, const int K,
             const double *A, const double *B, double *C)
{
  int i, j, k;

  for (i = 0; i < M; ++i) {
       const double *Ai_ = A + i;
       for (j = 0; j < N; ++j) {
            const double *B_j = B + j*lda;

            double cij = *(C + j*lda + i);

            for (k = 0; k < K; ++k) {
                 cij += *(Ai_ + k*lda) * *(B_j + k);
            }

            *(C + j*lda + i) = cij;
       }
  }
}

void
procesar_bloque (const int lda,
          const double *A, const double *B, double *C,
          const int i, const int j, const int k)
{
     /*
       Recordar que hay que tener en cuenta los bloques incompletos de los borde
s.
       Si la matriz es de 7x7 y los bloques de 3x3, hay bloques de 1x3, 3x1 y 1x
1.


            xxxoooX
            xxxoooX
            xxxoooX
            oooxxxO
            oooxxxO
            oooxxxO
            XXXOOOX

     */
     const int M = (i+TAM_BLOQUE > lda? lda-i : TAM_BLOQUE);
     const int N = (j+TAM_BLOQUE > lda? lda-j : TAM_BLOQUE);
```



```
     const int K = (k+TAM_BLOQUE > lda? lda-k : TAM_BLOQUE);

     matmult_basico (lda, M, N, K,
                     A + i + k*lda, B + k + j*lda, C + i + j*lda);
}

void
matmult_cuadrado (const int M,
                  const double *A, const double *B, double *C)
{
     const int n_blocks = M / TAM_BLOQUE + (M%TAM_BLOQUE? 1 : 0);
     int bi, bj, bk;

     for (bi = 0; bi < n_blocks; ++bi) {
          const int i = bi * TAM_BLOQUE;

          for (bj = 0; bj < n_blocks; ++bj) {
               const int j = bj * TAM_BLOQUE;

               for (bk = 0; bk < n_blocks; ++bk) {
                    const int k = bk * TAM_BLOQUE;

                    procesar_bloque (M, A, B, C, i, j, k);
               }
          }
     }
}
```

```
const char* matmult_desc = "Multiplicacion por columnas de A.";

void
matmult_cuadrado (const int M,
              const double *A, const double *B, double *C)
{
        int i,j,k;

        const double* Bi = B;
        double* Ci = C;

        for( i = 0; i < M; i++, Bi+=M, Ci+=M ) {
                const double* Aj = A;
                for( j = 0; j < M; j++, Aj+=M ) {
                        double Bij = Bi[j];
                        for( k = 0; k < M; k++ ) {
                                Ci[k] += Aj[k] * Bij;
                        }
                }
        }
}
```



```
const char* matmult_desc = "Multiplicacion de matrices por linea por bloques.";

/* Se puede jugar con este valor */
#if !defined(TAM_BLOQUE)
#define TAM_BLOQUE ((int) 95)
#endif

void
matmult_basico (const int lda,
             const int M, const int N, const int K,
             const double *A, const double *B, double *C)
{
        int i,j,k;

        const double* Bi = B;
        double* Ci = C;

        for( i = 0; i < N; i++, Bi+=lda, Ci+=lda ) {
                const double* Aj = A;
                for( j = 0; j < K; j++, Aj+=lda ) {
                        double Bij = Bi[j];
                        for( k = 0; k < M; k++ ) {
                                Ci[k] += Aj[k] * Bij;
                        }
                }
        }
}
#include <stdio.h>

void
procesar_bloque (const int lda,
          const double *A, const double *B, double *C,
          const int i, const int j, const int k)
{
     /*
       Recordar que hay que tener en cuenta los bloques incompletos de los borde
s.
       Si la matriz es de 7x7 y los bloques de 3x3, hay bloques de 1x3, 3x1 y 1x
1.


             xxxoooX
             xxxoooX
             xxxoooX
             oooxxxO
             oooxxxO
             oooxxxO
             XXXOOOX

     */
     const int M = (i+TAM_BLOQUE > lda? lda-i : TAM_BLOQUE);
     const int N = (j+TAM_BLOQUE > lda? lda-j : TAM_BLOQUE);
     const int K = (k+TAM_BLOQUE > lda? lda-k : TAM_BLOQUE);

     matmult_basico (lda, M, N, K,
                  A + i + k*lda, B + k + j*lda, C + i + j*lda);
}

void
matmult_cuadrado (const int M,
               const double *A, const double *B, double *C)
```

```
{
     const int n_blocks = M / TAM_BLOQUE + (M%TAM_BLOQUE? 1 : 0);
     int bi, bj, bk;

     for (bi = 0; bi < n_blocks; ++bi) {
          const int i = bi * TAM_BLOQUE;

          for (bj = 0; bj < n_blocks; ++bj) {
               const int j = bj * TAM_BLOQUE;

               for (bk = 0; bk < n_blocks; ++bk) {
                    const int k = bk * TAM_BLOQUE;

                    procesar_bloque (M, A, B, C, i, j, k);
               }
          }
     }
}
```



```
#include "timing.h"

#if defined(CLOCK_HIGHRES)
#define CLOCK CLOCK_HIGHRES
#elif defined(CLOCK_REALTIME)
#define CLOCK CLOCK_REALTIME
#else
#error No suitable clock found.  Check docs for clock_gettime.
#endif

long double
timespec_to_ldbl (struct timespec x)
{
  return x.tv_sec + 1.0E-9 * x.tv_nsec;
}

long double
timespec_diff (struct timespec start, struct timespec finish)
{
  long double out;
  out = finish.tv_nsec - (double)start.tv_nsec;
  out *= 1.0E-9L;
  out += finish.tv_sec - (double)start.tv_sec;
  return out;
}

long double
timer_resolution (void)
{
  struct timespec x;
  clock_getres (CLOCK, &x);
  return timespec_to_ldbl (x);
}

void
get_time (struct timespec* x)
{
  clock_gettime (CLOCK, x);
}
```

Table of Contents