Entregable 1

miércoles, 7 de abril de 2021 18:33

1. Resuelva el ejercicio 6 de la Práctica Nº 1 usando dos equipos diferentes: (1) cluster remoto y (2) equipo *hogareño* al cual tenga acceso (puede ser una PC de escritorio o una notebook).

6. Dada la ecuación cuadrática: $x^2 - 4.0000000\ x + 3.9999999 = 0$, sus raíces son $r_1 = 2.000316228$ y $r_2 = 1.999683772$ (empleando 10 dígitos para la parte decimal).

    a. El algoritmo *quadratic1.c* computa las raíces de esta ecuación empleando los tipos de datos *float* y *double*. Compile y ejecute el código. ¿Qué diferencia nota en el resultado?
    b. El algoritmo *quadratic2.c* computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución?
    c. El algoritmo *quadratic3.c* computa las raíces de esta ecuación, pero en forma repetida. Compile y ejecute el código variando la constante TIMES. ¿Qué diferencia nota en la ejecución? ¿Qué diferencias puede observar en el código con respecto a *quadratic2.c?*

    Nota: agregue el flag *-lm* al momento de compilar. Pruebe con y sin opciones de optimización del compilador.

Aclaración: todas las optimizaciones realizadas en este ejercicio van a ser del tipo O3

a) El tipo float solo utiliza 6 dígitos, por la cantidad de memoria que utiliza, por lo que redondea el número 3.9999999 en 4, mientras que double permite hasta 14 dígitos, utilizando más memoria. El redondeo del número que hace el float hace que el algoritmo con tipos float no calcule exactamente el resultado, como se puede apreciar abajo, la solución con Float da como resultado 2.00000 y 2.00000, mientras que los valores reales nos los da el double, con 2.00032 y 1.99968.
Sin optimizar:

```
Soluciones Float: 2.00000       2.00000
Soluciones Double: 2.00032      1.99968
spusuario15@frontend:~$
```

Optimizado:

```
Soluciones Float: 2.00000       2.00000
Soluciones Double: 2.00032      1.99968
spusuario15@frontend:~$
```

Valores resultado (no son tiempos):

Valores	Sin optimización	Optimizado
Float	2.00000 | 2.00000	2.00000 | 2.00000
Double	2.00032 | 1.99968	2.00032 | 1.99968

Análisis del algoritmo para ver la evolución de la ecuación:

Solución correcta (en doubles):

```
void dbl_solve(double a, double b, double c)
{
   double d = pow(b,2) - 4.0*a*c;
   // d = (-4)^2 - 4.0*1*3.9999999  = 0.000000399999999
   double sd = sqrt(d);
   // sd = 0.00063245553
   double r1 = (-b + sd) / (2.0*a);
   // r1 = (-(-4) + 0.00063245553) / (2.0*1) = 2.00031622776
```

```
   double r2 = (-b - sd) / (2.0*a);
   // r2 = (-(-4) - 0.00063245553) / (2.0*1) = 1.99968377224
   printf("Soluciones Double: %.5f\t%.5f\n", r1, r2);
}
```

Solución incorrecta (en floats):

```
void flt_solve(float a, float b, float c)
{
   float d = pow(b,2) - 4.0*a*c;
   // d = (-4)^2 - 4.0*1*4 = 0
   float sd = sqrt(d);
   // sd = 0
   float r1 = (-b + sd) / (2.0*a);
   // r1 = (-(-4) + 0) / (2.0*1) = 2
   float r2 = (-b - sd) / (2.0*a);
   // r2 = (-(-4) - 0) / (2.0*1) = 2
   printf("Soluciones Float: %.5f\t%.5f\n", r1, r2);
}
```

b) En el algoritmo quadratic2, como se puede apreciar, ante el programa sin optimizar, el calculo con floats tarda más. Esto no debería ser así, es decir, al float ocupar menos lugar, se supone que los bloques de memoria que se cachean deberían tener más números y por lo tanto el tiempo de ejecución menor. Pero si nos ponemos a analizar el algoritmo, podemos ver que las operaciones utilzadas esperan datos tipos double, por lo que el algoritmo float va a tener que realizar varias conversiones entre tipos, lo que va a empeorar su tiempo de ejecución, mientras que el algoritmo double, aunque debería tardar más, no necesita realizar las conversiones, por lo que termina siendo más rápido:

Ejecuciones en el cluster:
Sin optimizar:

TIME = 100

```
Tiempo requerido solucion Double: 45.119077
Tiempo requerido solucion Float: 46.865540
spusuario15@frontend:~$
```

TIME = 150

```
Tiempo requerido solucion Double: 67.680061
Tiempo requerido solucion Float: 70.227662
spusuario15@frontend:~$
```

TIME = 200

```
Tiempo requerido solucion Double: 90.205387
Tiempo requerido solucion Float: 93.713926
spusuario15@frontend:~$
```

Con el algoritmo optimizado con O3, el tiempo de la parte float es menor, ya que la optimización se encarga de mejorar el código para optimizar las conversiones, por lo que se cumple que el float al ocupar menos memoria se ejecuta más rápido:
Optimizado:

TIME = 100

```
Tiempo requerido solucion Double: 6.598324
Tiempo requerido solucion Float: 5.987267
spusuario15@frontend:~$
```

TIME = 150

```
Tiempo requerido solucion Double: 9.897131
Tiempo requerido solucion Float: 8.981394
spusuario15@frontend:~$
```

TIME = 200

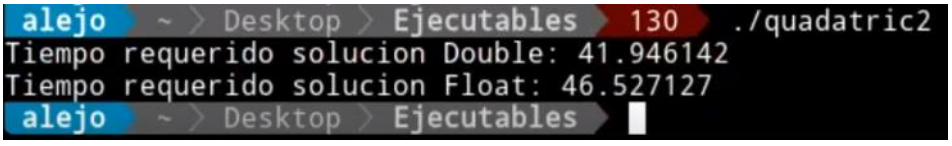
```
Tiempo requerido solucion Double: 13.006550
Tiempo requerido solucion Float: 11.988476
spusuario15@frontend:~$
```

Resumen:

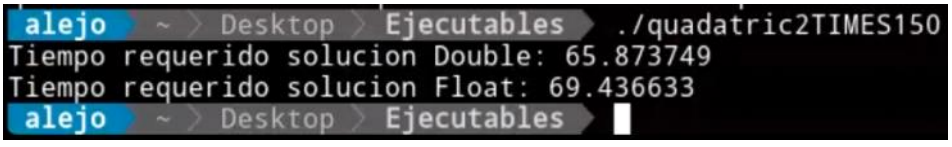
Tiempos	Float		Double	
	Sin optimizar	Optimizado	Sin optimizar	Optimizado
TIMES = 100	46.865540	5.987267	45.119077	6.598324
TIMES = 150	70.227662	8.981394	67.680061	9.897131
TIMES = 200	93.713926	11.988476	90.205387	13.006550

Como se puede apreciar en la tabla, sin optimzar la solución en doubles es más rápida, mientras que optimizada la solución en floats es más rápida.

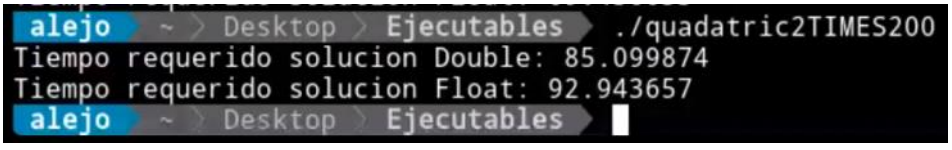
Ejecuciones en el equipo hogareño:
Sin optimizar:
TIME = 100



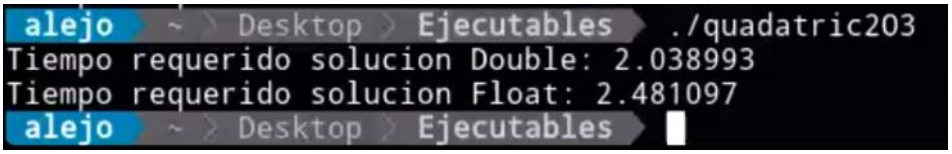
TIME = 150



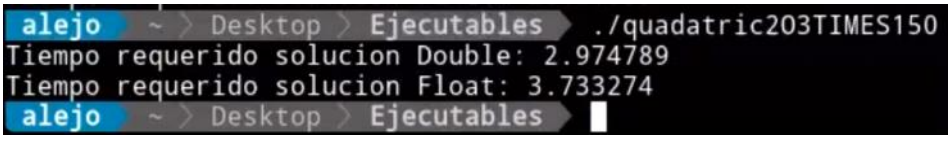
TIME = 200



Optimizado:
TIME = 100



TIME = 150



TIME = 200

```
alejo ~ Desktop Ejecutables ./quadatric2O3TIMES200
Tiempo requerido solucion Double: 3.935589
Tiempo requerido solucion Float: 5.060273
alejo ~ Desktop Ejecutables
```

Resumen:

Tiempos	Float		Double	
	Sin optimizar	Optimizado	Sin optimizar	Optimizado
TIMES = 100	46.527127	2.481097	41.946142	2.038993
TIMES = 150	69.436633	3.733274	65.873749	2.974789
TIMES = 200	92.943657	5.060273	85.099874	3.935589

En este caso las soluciones con doubles son más rápidas, pero sospechamos que se debe a que las arquitecturas x64 favorecen a este tipo de datos.

Ejemplos de las conversiones:
Solución en float:

```
for (j=0; j<TIMES ; j++)
        for (i=0; i<N ; i++) {
            //flt_solve(fa[i], fb[i], fc[i]);
            float d = pow(fb[i],2) - 4.0*fa[i]*fc[i];
            /*d = double - double * float * float | cabe destacar también que pow espera un
            double, y se le pasa un float, lo que necesita otra conversión */
            float sd = sqrt(d);
            // con sqrt pasa lo mismo que con pow, debe transformar el float que le llega a
            double
            float r1 = (-fb[i] + sd) / (2.0*fa[i]);
            //-float + float / double * float
            float r2 = (-fb[i] - sd) / (2.0*fa[i]);
            //-float - float / double * float
        }
```

En el caso de la solución con doubles, estos problemas no se dan, ya que todas las operaciones se hacen entre doubles y las funciones también esperan doubles.

c) En este algoritmo la solución en float no requiere las conversiones, ya que las funciones (como powf y sqtrf) reciben datos del tipo float, entonces las operaciones también se hacen en float, lo que debería mejorar el rendimiento.
A su vez también hay que tener en cuenta que la solución en float utiliza en la función powf un "2.0f", es decir, un número en punto flotante, lo que empeora el rendimiento en comparación a la solución en double, que tiene una constante al hacer el exponente. En el cluster, la versión sin optimizar del algoritmo float es peor, pero en el equipo hogareño el algoritmo float es mejor tanto optimizado como sin optimizar, esta diferencia probablemente se deba a la diferencia en las arquitecturas de ambos equipos.

Ejecuciones en el cluster:
Sin optimización:
TIMES = 100

```
Tiempo requerido solucion Double: 45.127147
Tiempo requerido solucion Float: 69.127278
spusuario15@frontend:~$
```

TIMES = 150

```
Tiempo requerido solucion Double: 67.634059
Tiempo requerido solucion Float: 103.696443
spusuario15@frontend:~$
```

TIMES = 200

```
Tiempo requerido solucion Double: 90.240597
Tiempo requerido solucion Float: 138.277366
spusuario15@frontend:~$
```

Optimizado:
TIMES = 100

```
Tiempo requerido solucion Double: 6.534946
Tiempo requerido solucion Float: 3.408966
spusuario15@frontend:~$
```

TIMES = 150

```
Tiempo requerido solucion Double: 9.840311
Tiempo requerido solucion Float: 5.111118
spusuario15@frontend:~$
```

TIMES = 200

```
Tiempo requerido solucion Double: 13.096166
Tiempo requerido solucion Float: 6.850635
spusuario15@frontend:~$
```

Resumen:

Tiempos	Float		Double	
	Sin optimizar	Optimizado	Sin optimizar	Optimizado
TIMES = 100	69.127278	3.408966	45.127147	6.534946
TIMES = 150	103.696443	5.111118	67.634059	9.840311
TIMES = 200	138.277366	6.850635	90.240597	13.096166

Ejecuciones en el equipo hogareño:
Sin optimización:
TIMES = 100

```
alejo ~ > Desktop > Ejecutables ./quadatric3
Tiempo requerido solucion Double: 42.006967
Tiempo requerido solucion Float: 29.751669
alejo ~ > Desktop > Ejecutables
```

TIMES = 150

```
alejo ~ > Desktop > Ejecutables ./quadatric3TIMES150
Tiempo requerido solucion Double: 62.881591
Tiempo requerido solucion Float: 43.043976
alejo ~ > Desktop > Ejecutables
```

TIMES = 200

```
alejo ~ > Desktop > Ejecutables ./quadatric3TIMES200
Tiempo requerido solucion Double: 85.530008
Tiempo requerido solucion Float: 57.355605
alejo ~ > Desktop > Ejecutables
```

Optimizado:
TIMES = 100

```
alejo ~ Desktop Ejecutables ./quadatric303
Tiempo requerido solucion Double: 1.943549
Tiempo requerido solucion Float: 1.503209
alejo ~ Desktop Ejecutables
```

TIMES = 150

```
alejo ~ Desktop Ejecutables ./quadatric303TIMES150
Tiempo requerido solucion Double: 2.910247
Tiempo requerido solucion Float: 2.147763
alejo ~ Desktop Ejecutables
```

TIMES = 200

```
alejo ~ Desktop Ejecutables ./quadatric303TIMES200
Tiempo requerido solucion Double: 3.858832
Tiempo requerido solucion Float: 2.918814
alejo ~ Desktop Ejecutables
```

Resumen:

Tiempos	Float		Double	
	Sin optimizar	Optimizado	Sin optimizar	Optimizado
TIMES = 100	29.751669	1.503209	42.006967	1.943549
TIMES = 150	43.043976	2.147763	62.881591	2.910247
TIMES = 200	57.355605	2.918814	85.530008	3.858832

2. Desarrolle un algoritmo en el lenguaje C que compute la siguiente ecuación:

$$C = T + avg_R(RA + RB)$$

Donde A, B, C, T y R son matrices cuadradas de NxN.
$avgR$ es el valor promedio de los elementos de la matriz R.
El elemento (i,j) de la matriz R debe calcularse como:

$$R_{i,j} = (1 - T_{i,j})(1 - cos\theta_{i,j}) + T_{i,j} sin\theta_{i,j}$$

Donde $T_{i,j}$ es el elemento en la posición (i,j) de la matriz T.
El angulo $\theta_{i,j}$, en radianes, se obtiene de la posición (i,j) de una matriz M de NxN. Los valores de los elementos de la matriz M están comprendidos en un rango entre 0 y 2π.

Mida el tiempo de ejecución del algoritmo en el cluster remoto. Las pruebas deben considerar la variación del tamaño de problema ($N=\{512, 1024, 2048, 4096\}$). Por último, recuerde aplicar las técnicas de programación y optimización vistas en clase.

Explicación:

A la hora de leer el problema, identificamos que tiene un gran uso de matrices, por lo que nos concentramos en optimizar el código para su funcionamiento. Vamos a hacer un recorrido del código explicando paso a paso las decisiones tomadas. Por cada decisión que tomamos, vamos a hacer un cuadro comparativo entre el tiempo del código si no se tomaran las decisiones buenas y el tiempo del código con todas las mejores decisiones (el archivo adjunto en la entrega). Aclarar antes de empezar que las ejecuciones están hechas con el código con optimización del compilador O3 y, cuando participe la multiplicación de matrices, con un tamaño de bloque de 64 que, como mostramos en la tabla que se encuentra en "decisión 2", es el óptimo.

Decisión 1:
A la hora de la declaración de las matrices, utilizamos la forma de "arreglo dinámico como vector de elementos", alternativa 4 de la segunda teoría, ya que es la forma más efectiva para utilizar matrices, tiene todos sus datos contiguos en memoria, nos permite elegir si organizar los datos por filas o por columnas (en nuestro caso optamos por la opción "por filas"), no tiene un tamaño máximo por la forma en que está declarado, su tamaño máximo está definido por la memoria del sistema y, además, puede cambiar su tamaño en ejecución.
La otra forma de hacer la alocación es con la técnica "arreglo dinámico como vector de punteros a filas/columnas", la desventaja de la misma es que los datos no se alocan de forma continua en memoria, y no se puede aprovechar la localidad de datos ya que, por ejemplo, no se puede realizar la multiplicación de matrices por bloques.

Usando la técnica "arreglo dinámico como vector de punteros a filas/columnas":

```
//Alocacion con la tecnica arreglo dinámico como vector de punteros a filas/columnas
A=malloc(N*sizeof(double*));
B=malloc(N*sizeof(double*));
C=malloc(N*sizeof(double*));
T=malloc(N*sizeof(double*));
R=malloc(N*sizeof(double*));
M=malloc(N*sizeof(double*));
RA=malloc(N*sizeof(double*));
RB=malloc(N*sizeof(double*));
for (int i = 0; i < N; i++) {
    A[i] = malloc(N*sizeof(double*));
    B[i] = malloc(N*sizeof(double*));
    C[i] = malloc(N*sizeof(double*));
    T[i] = malloc(N*sizeof(double*));
    R[i] = malloc(N*sizeof(double*));
    M[i] = malloc(N*sizeof(double*));
    RA[i] = malloc(N*sizeof(double*));
    RB[i] = malloc(N*sizeof(double*));
}
```

Usando la técnica "arreglo dinámico como vector de elementos":

```
//Alocacion del espacio de memoria siguiendo la tecnica arreglo dinamico como vector de elementos
A=(double *) malloc(N*N*sizeof(double));
B=(double *) malloc(N*N*sizeof(double));
C=(double *) malloc(N*N*sizeof(double));
T=(double *) malloc(N*N*sizeof(double));
R=(double *) malloc(N*N*sizeof(double));
M=(double *) malloc(N*N*sizeof(double));
RA=(double *) malloc(N*N*sizeof(double));
RB=(double *) malloc(N*N*sizeof(double));
```

Comparación de eficiencia (recordar que la multiplicación de matrices es la simple cuando se usa la técnica "arreglo dinámico como vector de punteros a filas/columnas"):

N	Sin tomar la decisión ("arreglo dinámico como vector de punteros a filas/columnas")	Tomando la decisión ("arreglo dinámico como vector de elementos")*
512	0.360764	0.353109
1024	5.263891	4.983425
2048	41.331313	39.806705
4096	322.971997	319.719473

**multiplicación de matrices simple*

Como se puede notar, la alocación con la técnica de "arreglo dinámico como vector de elementos" es mejor por su localidad de datos, pero la diferencia no es tan amplia. Sin embargo, la técnica que usa el vector de elementos es ampliamente mejor, ya que nos permite utilizar la multiplicación por bloques, que da un gran salto en los tiempos de ejecución:

N (tamaño de bloque = 64)	Sin tomar la decisión ("arreglo dinámico como vector de punteros a filas/columnas")	Tomando la decisión ("arreglo dinámico como vector de elementos " y multiplicación por bloques)
512	0.360764	0.555820
1024	5.263891	4.327450
2048	41.331313	34.393347
4096	322.971997	268.753957

Con el tamaño de bloque 512, la alternativa con multiplicación por bloques es ligeramente peor, pero esto lo explicamos más adelante en la decisión 3.

Para la inicialización de las matrices utilizamos números randoms entre 1 y 100.
Luego de la inicialización empieza el cómputo de las matrices, por lo que inicializamos el contador de tiempo de ejecución.
El primer paso es calcular la matriz R. A la hora de calcular los valores, simplemente tuvimos que hacer algunos calculos y acceder a los datos de otras matrices, por lo que en una línea se puede calcular el valor de una posición de R.
Metimos el cálculo dentro de dos for's, para que se recorra cada posición de la matriz y se complete.

La acción siguiente es calcular el promedio de R, además de su posterior multiplicación con la matriz A y B.

Decisión 2:
Para el promedio de R, aprovechamos los for's que se habían definido anteriormente. Como en los mismos se recorría R, los utilizamos para ir llevando la sumatoria de valores que luego nos definen el promedio.
Otra alternativa para el cálculo del promedio es realizar la sumatoria después del cálculo de R, escribiendo otros dos for's para volver a iterar sobre la matriz.
Claramente la forma que elegimos utilizar es más eficiente, ya que nos ahorramos volver a recorrer R después de calcular sus valores.
Teniendo la sumatoria, solo queda dividir la misma por N*N y obtenemos el promedio.

Código con la sumatoria separada:

```
//Calculo de la matriz R
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        R[i*N+j] = (1 - T[i*N+j])*(1- cos(M[i*N+j])) + T[i*N+j]*sin(M[i*N+j]);
    }
}

//Calculo de la sumatoria de R
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        sumatoria = R[i*N+j] + sumatoria;
    }
}
//Calculo del promedio de R
promedioR = sumatoria / (N*N);
```

Código con la sumatoria optimizada:

```
//Calculo de la matriz R
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        R[i*N+j] = (1 - T[i*N+j])*(1- cos(M[i*N+j])) + T[i*N+j]*sin(M[i*N+j]);
        sumatoria = R[i*N+j] + sumatoria; // Calculo utilizado posteriormente para el promedio de R
    }
}

//Calculo del promedio de R
promedioR = sumatoria / (N*N);
```

Comparación de eficiencia:

N (tamaño de bloque = 64)	Sin tomar la decisión (calculo de sumatoria aparte)	Tomando la decisión (calculo de sumatoria aprovechando el for anterior)
512	0.557915	0.555820
1024	4.340003	4.327450
2048	34.433580	34.393347
4096	269.703397	268.753957

Como se puede ver, la sumatoria optimizada es ligeramente mejor.

Decisión 3:
Lo próximo a realizar es la multiplicación de R con A y con B.
Para realizar la multiplicación de matrices se pueden utilizar varias técnicas, como los datos están contiguos en memoria, ya que alocamos con la técnica "arreglo dinámico como vector de elementos", podemos elegir la forma que queramos. La más intuitiva y simple es hacer tres for's anidados, en donde los dos primeros recorren cada posición de la matriz resultante, y el tercero sirve para recorrer la fila y la columna de las variables que se multiplican.
Si bien es la solución más simple, hay otra solución más eficiente, la multiplicación por bloques, que es la nosotros utilizamos.
La multiplicación en bloques propone aprovechar los bloques de memoria que se llevan a cache, en lugar de calcular celda a celda la multiplicación, se calcula bloque a bloque.
También se utiliza el triple for, pero con la diferencia de que no se multiplica una celda por cada iteración, sino que se multiplican todas las celdas que entren en el bloque indicado (el cual debería ser lo más ajustado al de la caché posible). De esta manera, estamos haciendo una utilización óptima de la cache, accediendo menos veces a memoria para leer y escribir los valores de las matrices en juego.
Para calcular el tamaño óptimo del bloque tenemos que tener en cuenta dos cosas, el tamaño de nuestra caché, y el tamaño de los valores de la matriz, no es lo mismo usar floats, por ejemplo, que doubles. La idea es que entren en la caché las tres matrices, la mayor cantidad de valores que se pueda.
Testeo sobre diferentes tamaños de bloques (con el mejor código):

N	Tamaño de bloque			
	16	32	64	128
512	0.669971	0.588469	0.557806	0.565759
1024	5.209323	4.551201	4.342536	4.393844
2048	41.164502	36.021259	34.283854	34.954683
4096	321.639836	282.722674	268.846921	273.874459

Como se puede ver, el tamaño de bloque óptimo es 64.

Código con multiplicación básica:

```
// Multiplicacion de R con A y R con B
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int k = 0; k < N; k++) {
            RA[i*N+j] += R[i*N+k] * A[j*N+k];
            RB[i*N+j] += R[i*N+k] * B[j*N+k];
        }
    }
}
```

Código con multipliación por bloques:

```
//Multiplicacion de R con A
matmulblks(R, A, RA, N, bs);
//Multiplicacion de R con B
matmulblks(R, B, RB, N, bs);
```

(...)

```
void matmulblks(double *a, double *b, double *c, int n, int bs) {
    int i, j, k;

    for (i = 0; i < n; i += bs) {
        for (j = 0; j < n; j += bs) {
            for  (k = 0; k < n; k += bs) {
                blkmul(&a[i*n + k], &b[j*n + k], &c[i*n + j], n, bs);
            }
        }
    }
}

void blkmul(double *ablk, double *bblk, double *cblk, int n, int bs) {
    int i, j, k;

    for (i = 0; i < bs; i++) {
        for (j = 0; j < bs; j++) {
            for  (k = 0; k < bs; k++) {
                cblk[i*n + j] += ablk[i*n + k] * bblk[j*n + k];
            }
        }
    }
}
```

Comparación de eficiencia:

N (tamaño de bloque = 64)	Sin tomar la decisión (multiplicación de matrices simple)	Tomando la decisión (multiplicación de matrices por bloques)
512	0.353109	0.555820
1024	4.983425	4.327450
2048	39.806705	34.393347
4096	319.719473	268.753957

Como se puede ver en el cuadro, para valores bajos la multiplicación simple es más rápida, ya que el overhead generado por llamar a 2 funciones y hacer algunos for's más que se realizan en la multiplicación por bloques es mayor que el beneficio que genera la técnica. Sin embargo, se aprecia como rápidamente la multiplicación de bloques se hace mucho más efectiva mientras sube el tamaño de la matriz.

Decisión 4:
Por último queda calcular C.
La forma es similar a la utilizada en R, se hace un doble for para recorrer cada posición de C e ir calculado sus valores.
Como la ecuación no es demasiado compleja, se puede hacer todo en una línea, que es la manera más óptima, pero si se deseara también se podrían hacer varios "dobles for", en donde, por ejemplo, el primero calcula (RA + RB) * avgR y lo guarda en otra matriz auxiliar, y después se podría hacer otro para multiplicar esa matriz auxiliar por el promedio de R, pero sería un desperdicio de cómputo innecesario, ya que con un doble for se puede calcular tranquilamente.

Versión con código separado:

```
//Calculo de C
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        AVRXAB[i*N+j] = ((RA[i*N+j] + RB[i*N+j]) * promedioR);
    }
}

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i*N+j] = T[i*N+j] + AVRXAB[i*N+j];
    }
}
```

Versión con código junto:

```
//Calculo de C
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        C[i*N+j] = T[i*N+j] + ((RA[i*N+j] + RB[i*N+j]) * promedioR);
    }
}
```

Comparación de eficiencia:

N (tamaño de bloque = 64)	Sin tomar la decisión (cálculo separado)	Tomando la decisión (cálculo junto)
512	0.559171	0.555820
1024	4.342561	4.327450
2048	34.443319	34.393347
4096	269.099964	268.753957

Como se puede ver en el cuadro, calcular las cosas en un solo for es ligeramente mejor

Luego del cálculo de C solo nos queda imprimir en pantalla el tiempo de ejecución, y luego de eso liberar la memoria ocupada por las matrices.

Datos del equipo hogareño:
Datos del HW:
Procesador: Intel Core i7-4702MQ 2.20GHz
Memoria RAM: 16 GB
Tipo de sistema: sistema operativo de 64 bits, procesador x64
Disco: 223GB KINGSTON SHFS37A240G (SATA-3 (SSD))
Placa de video: Intel HD Graphics 4600 (CLEVO/KAPOK Computer)

Datos del SW:
Distribución de linux: Linux Lite 5.4 64bit - 1st April, 2021