2º curso / 2º cuatr.

Grado Ing. Inform.

Arquitectura de Computadores (AC)

Cuaderno de prácticas.
Bloque Práctico 5. Optimización de código

Estudiante (nombre y apellidos): David Martínez Díaz
Grupo de prácticas y profesor de prácticas: 2 – Juan José Escobar
Fecha de entrega: 01/06/2021
Fecha evaluación en clase: 03/06/2021

Antes de comenzar a realizar el trabajo de este cuaderno consultar el fichero con los normas de prácticas que se encuentra en SWAD

Denominación de marca del chip de procesamiento o procesador (se encuentra en /proc/cpuinfo): *Intel(R) Xeon(R) CPU E3-1230 v6 @ 3.50GHz*

Sistema operativo utilizado: *(Linux version 3.10.0-957.27.2.el7.x86_64)*

Versión de gcc utilizada: *(gcc version 4.8.5 20150623)*

Volcado de pantalla que muestre lo que devuelve `lscpu` en la máquina en la que ha tomado las medidas:

```
[e2estudiante14@atcgrid bp4]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 158
Model name:            Intel(R) Xeon(R) CPU E3-1230 v6 @ 3.50GHz
Stepping:              9
CPU MHz:               1600.036
CPU max MHz:           3900.0000
CPU min MHz:           800.0000
BogoMIPS:              7008.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

1. **(a)** Implementar un código secuencial que calcule la multiplicación de dos matrices cuadradas. Utilizar como base el código de suma de vectores de BP0. Los datos se deben generar de forma aleatoria para un número de filas mayor que 8, como en el ejemplo de BP0, se puede usar `drand48()`.

   MULTIPLICACIÓN DE MATRICES:

   CAPTURA CÓDIGO FUENTE: `pmm-secuencial.c`

```c
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h>  // biblioteca donde se encuentra la funciÃ³n printf()
#include <time.h>   // biblioteca donde se encuentra la funciÃ³n clock_gettime()


//#define VECTOR_LOCAL  // descomentar para que los vectores sean variables ...
//#define VECTOR_GLOBAL // descomentar para que los vectores sean variables ...
#define VECTOR_DYNAMIC  // descomentar para que los vectores sean variables ...


#ifdef VECTOR_GLOBAL
#define MAX 33554432    //=2^25
//#define MAX 4294967295  //=2^32 -1

double m1[MAX], m2[MAX], m3[MAX];
#endif

int main(int argc, char** argv){

  int i, j, suma;

  struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecuciÃ³n

  //Leer argumento de entrada (nÂº de componentes del vector)
  if (argc<2){
    printf("Faltan nÂº componentes del vector\n");
    exit(-1);
  }

  unsigned int N = atoi(argv[1]);
  #ifdef VECTOR_LOCAL
  double m1[N], m2[N], m3[N];   // TamaÃ±o variable local en tiempo de ejecuciÃ³n ...
                                // disponible en C a partir de C99
  #endif

  #ifdef VECTOR_GLOBAL
  if (N>MAX) N=MAX;
```

```c
// Definir las matrices
  #ifdef VECTOR_DYNAMIC

  int **m1, **m2, **m3;
  m1 = (int**) malloc(N*sizeof(int*));
  m2 = (int**) malloc(N*sizeof(int*));
  m3 = (int**) malloc(N*sizeof(int*));

  for(i=0; i<N; i++){
      m1[i] = (int *) malloc(N*sizeof(int));
      m2[i] = (int *) malloc(N*sizeof(int));
      m3[i] = (int *) malloc(N*sizeof(int));
  }


    if ( m1==NULL || m2==NULL || m3==NULL ){
        printf("Error en la reserva de espacio para la matriz");
        exit(-2);

    }
  #endif

  //Inicializar matrices
  if (N>8){

        for(i=0; i<N; i++){
            for(j=0; j<N; j++){
                m1[i][j] = drand48();
                m2[i][j] = drand48();
                m3[i][j] = 0;
            }
        }
  }
```

```c
  else {

      for(i=0; i<N; i++){
          for(j=0; j<N; j++){
              m1[i][j] = N+i+2;
              m2[i][j] = N+i+4;
              m3[i][j] = 0;
          }
      }

  }
clock_gettime(CLOCK_REALTIME,&cgt1);

// Calcular multiplicacion de matrices

for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        m3[i][j] = m1[i][j]*m2[i][j];
    }
}

// Suma total
for(i=0; i<N; i++)
    for(j=0; j<N; j++)
        suma += m3[i][j];

clock_gettime(CLOCK_REALTIME,&cgt2);
ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
     S  (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));

  //Imprimir resultado de la multiplicacion y el tiempo de ejecucion

  printf("Tiempo:%11.9f\t / Tamanio Matriz:%u / Resultado Matriz:%u \n",ncgt,N, suma);
```

```c
  #ifdef VECTOR_DYNAMIC

    for(i=0; i<N; i++){
        free(m1[i]);
        free(m2[i]);
        free(m3[i]);
    }

    free(m1); // libera el espacio reservado para v1
    free(m2); // libera el espacio reservado para v2
    free(m3); // libera el espacio reservado para v3

  #endif
  return 0;
}
```

(b) Modificar el código (solo el trozo que calcula la multiplicación) para reducir el tiempo de ejecución. Justificar los tiempos obtenidos (usando siempre -O2) a partir de la modificación realizada. Incorporar los códigos modificados en el cuaderno.

MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación A) –explicación-: Para optimizar los accesos a la memoria, en mi caso, de la segunda matriz, lo hacemos cogiendo las columnas no haciendo uso de la localidad del acceso respectivo. Sin embargo, si realizamos una trasposición se soluciona el problema. Es decir, nos ahorramos tener que acceder a posiciones de memoria que no se hallan dentro del cache.

Modificación B) –explicación-: Para este caso he realizado un desenrollado de la matriz para romper secuencias de instrucciones dependientes intercalando otras instrucciones, reduciendo el número de saltos.

...

CÓDIGOS FUENTE MODIFICACIONES

A) Captura de `pmm-secuencial-modificado_A.c`

```
88  // Metodo de optimizacion: trasponemos la matriz M2 para conseguir accesos secuencia
89
90  int tmp[N][N];
91
92  for(i=0; i<N; i++){
93      for(j=0; j<N; j++){
94          tmp[i][j] = m2[j][i];
95      }
96  }
97
98  clock_gettime(CLOCK_REALTIME,&cgt1);
99
100     for (i = 0; i < N; i++){
101         for (j = 0; j < N; ++j){
102             for (k = 0; k < N; ++k)
103                 suma += m1[i][k] * tmp[j][k];
104
105         m3[i][j] = suma;
106         suma = 0;
107     }
108 }
```

Capturas de pantalla (que muestren la compilación y que el resultado es correcto):

```
[e2estudiante14@atcgrid ejer1]$ gcc -O2 pmm-secuencial-modificado_A.c
 -o pmm-secuencial-modificado_A -lrt
[e2estudiante14@atcgrid ejer1]$ srun -p ac pmm-secuencial-modificado_
A 1000
Tiempo:1.300088890       / Tamanio Matriz:1000 / Resultado Matriz:187
6956988
```

B) Captura de `pmm-secuencial-modificado_B.c`

```
// Metodo de optimizacion: desenrrollado de la matriz

int r = N/4;

for (i = 0; i < N; i++){
    for (j = 0; j < N; j+=4){
        for (k = 0; k < N; ++k){

            m3[i][j] += m1[i][k] * m2[k][j];
            m3[i][j+1] += m1[i][k] * m2[k][j];
            m3[i][j+2] += m1[i][k] * m2[k][j];
            m3[i][j+3] += m1[i][k] * m2[k][j];
        }
    }
}

for (i = 0; i < N; i++)
    for (j = N-r; j < N; j++)
        for (k = 0; k < N; ++k)
            m3[i][j] += m1[i][k] * m2[k][j];
```

Capturas de pantalla (que muestren la compilación y que el resultado es correcto):

```
[e2estudiante14@atcgrid ejer1]$ gcc -O2 pmm-secuencial-modificado_B.c
 -o pmm-secuencial-modificado_B -lrt
[e2estudiante14@atcgrid ejer1]$ srun -p ac pmm-secuencial-modificado_
B 1000
Tiempo:4.833309190       / Tamanio Matriz:1000 / Resultado Matriz:326
0763776
[e2estudiante14@atcgrid ejer1]$
```

TIEMPOS:

Modificación	Breve descripción de las modificaciones	-O2
Sin modificar	*No hay modificaciones en este código*	9.778395409
Modificación A)	*Hemos traspuesto la M2 para conseguir un acceso secuencial en la cache.*	1.300088890
Modificación B)	*Desenrollado de la matriz para reducir el número de saltos*	4.833309190
...		

COMENTARIOS SOBRE LOS RESULTADOS Y JUSTIFICACIÓN DE LAS MEJORAS EN TIEMPO:

Según los resultados obtenidos, desenrollar el bucle hace un gran impacto sobre la optimización con un tiempo de 4,83 segundos, porque asi conseguimos en menos iteraciones realizar más operaciones, teniendo una reducción de saltos y que haya una amplitud de instrucciones independientes.

Pero, por otro lado, lo más eficiente es utilizar una matriz traspuesta, nuestro caso M2, consiguiendo una mejora en el acceso de datos ya que la cache se coge en las filas, con un tiempo de 1,30 segundos.

2. **(a)** Usando como base el código de BP0, generar un programa para evaluar un código de la Figura 1. M y N deben ser parámetros de entrada al programa. Los datos se deben generar de forma aleatoria para valores de M y N mayores que 8, como en el ejemplo de BP0.

CÓDIGO FIGURA 1:

CAPTURA CÓDIGO FUENTE: figura1-original.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char** argv){

  int i, j, k, suma;
  struct timespec cgt1,cgt2; double ncgt;

  if (argc<3){
    printf("Faltan n componentes del vector\n");
    exit(-1);
  }

  unsigned int N = atoi(argv[1]);
  unsigned int M = atoi(argv[2]);

  struct {
    int a;
    int b;
  }s[N];

    int *R;
    int X1, X2;

    R = (int*) malloc(N*sizeof(int));

  //Inicializar vectores
  if (N > 8){

    srand(time(0));
    for (i = 0; i < N; i++){

        s[i].a = rand();
        s[i].b = rand();
        R[i] = 0;
    }
  }
  else{

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

        s[i].a =  N+i+2;
        s[i].b = N+i+4;
        R[i] = 0;
    }
  }

  clock_gettime(CLOCK_REALTIME,&cgt1);

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

        X1=0; X2=0;
        for(j=0; j<N; j++)
            X1+=2*s[j].a+i;

        for(k=0; k<N; k++)
            X2+=3*s[k].b-i;

        if (X1<X2){
            R[i]=X1;
        }

        else{
            R[i]=X2;
        }
   }

  clock_gettime(CLOCK_REALTIME,&cgt2);

  //Calcular suma de vectores
  for(i=0; i<N; i++)
    suma += R[i];

  ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
       (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));

  //Imprimir resultado de la suma y el tiempo de ejecucion
  printf("Tiempo:%11.9f\t / Tamanio vector:%u / Resultado Vector:%u \n",ncgt,N, suma);

  return 0;
}
```

Figura 1 . Código C++ que suma dos vectores. M y N deben ser parámetros de entrada al programa, usar valores mayores que 1000 en la evaluación.

```
struct {
        int a;
        int b;
}  s[N];

main()
{
   ...
   for (ii=0; ii<M;ii++) {
      X1=0; X2=0;
```

```
      for(i=0; i<N;i++)  X1+=2*s[i].a+ii;
      for(i=0; i<N;i++)  X2+=3*s[i].b-ii;

      if (X1<X2)  R[ii]=X1  else  R[ii]=X2;
   }
   ...
}
```

(b) Modificar el código C (solo el trozo a evaluar) para reducir el tiempo de ejecución. Justificar los tiempos obtenidos (usando siempre -O2) a partir de la modificación realizada. En las ejecuciones de evaluación usar valores de N y M mayores que 1000. Incorporar los códigos modificados en el cuaderno.

MODIFICACIONES REALIZADAS (al menos dos modificaciones):

Modificación A) –explicación-: Para este caso he decidido realizar una localidad de accesos, por ello dependiendo de cómo se declaren los vectores va a decirnos como se almacena en memoria, por lo que podemos ahorrarnos un bucle for y juntar las dos sumatorias optimizándolo de esa manera.

Modificación B) –explicación-: En este apartado he decidido cambiar la estructura del struct para poder coger de manera más eficiente el acceso a memoria, ya que este se guarda de manera más eficiente en la memoria cache permitiendo conseguir unos tiempos más rápidos.

...

CÓDIGOS FUENTE MODIFICACIONES

A) Captura figura1-modificado_A.c

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

    X1=0; X2=0;

    for(j=0; j<N; j++){
        X1+=2*s[j].a+i;
        X2+=3*s[j].b-i;
    }

    if (X1<X2){
        R[i]=X1;
    }

    else{
        R[i]=X2;
    }
}
```

Capturas de pantalla (que muestren la compilación y que el resultado es correcto):

```
[e2estudiante14@atcgrid ejer2]$ srun -p ac figura1-modific
ado_A 50000 50000
Tiempo:3.104167246       / Tamanio vector:50000 / Resultad
o Vector:3378960172
[e2estudiante14@atcgrid ejer2]$
```

B) Captura figura1-modificado_B.c

```
18  struct {
19    int a[N];
20    int b[N];
21  }s;
22    int *R;
23    int X1, X2;
24    R = (int*) malloc(N*sizeof(int));
25
26  //Inicializar vectores
27  if (N > 8){
28
29    srand(time(0));
30    for (i = 0; i < N; i++){
31
32        s.a[i] = rand();
33        s.b[i] = rand();
34        R[i] = 0;
35    }
36  }
37
38  else{
39
40    for (i = 0; i < N; i++){
41
42        s.a[i] =  N+i+2;
43        s.b[i] = N+i+4;
44        R[i] = 0;
45    }
46  }
47
48  clock_gettime(CLOCK_REALTIME,&cgt1);
49
```

```
50  for (i=0; i<M; i++) {
51
52        X1=0; X2=0;
53        for(j=0; j<N; j++)
54            X1+=2*s.a[j]+i;
55
56        for(k=0; k<N; k++)
57            X2+=3*s.b[k]-i;
58
59        if (X1<X2){
60            R[i]=X1;
61        }
62
63        else{
64            R[i]=X2;
65        }
66   }
```

Capturas de pantalla (que muestren la compilación y que el resultado es correcto):

```
[e2estudiante14@atcgrid ejer2]$ srun -p ac figura1-modific
ado_B 50000 50000
Tiempo:4.855956956       / Tamanio vector:50000 / Resultad
o Vector:3790868785
[e2estudiante14@atcgrid ejer2]$
```

TIEMPOS:

Modificación	**Breve descripción de las modificaciones**	**-O2**
Sin modificar	*Sin modificaciones*	*4.913615948*
Modificación A)	*Localidad de accesos, ahorrándonos un bucle for*	*3.104167246*
Modificación B)	*Cambio de la variable struct s*	*4.855956956*
...		

COMENTARIOS SOBRE LOS RESULTADOS Y JUSTIFICACIÓN DE LAS MEJORAS EN TIEMPO:

Como conclusiones podemos decir que en primer lugar sin cambiar el código original pero cambiando la estructura del struct, es decir, la forma en el que guardamos los datos podemos conseguir una ligera eficiencia, pero casi imperceptible ya que son unos cuantos milisegundos, pero es una buena forma de mejorar la localidad de datos.

Por otro lado, cuando juntamos los dos bucles para hacer la sumatoria se produce una mejora importante, bastante más eficiente ya que al ahorrarnos un bucle for entero conseguir optimizar el código en casi 2 segundos.

En conclusión, nos sale más eficiente el ahorrarnos un bucle for que mejorar la localidad de datos.

3. El benchmark Linpack ha sido uno de los programas más ampliamente utilizados para evaluar las prestaciones de los computadores. De hecho, se utiliza como base en la lista de los 500 computadores más rápidos del mundo (el Top500 Report). El núcleo de este programa es una rutina que opera con flotantes de doble precisión denominada DAXPY (***D**ouble precision- real **A**lpha **X** **P**lus **Y***) que multiplica un vector por una constante y los suma a otro vector (Lección 3/Tema 1):

```
for (i=0;i<N;i++) y[i]= a*x[i] + y[i];
```

Generar los programas en ensamblador para cada una de las siguientes opciones de optimización del compilador: -O0, -Os, -O2, -O3. Explique las diferencias que se observan en el código justificando al mismo tiempo las mejoras en velocidad que acarrean. Incorporar los códigos al cuaderno de prácticas y destacar las diferencias entre ellos. Sólo se debe evaluar el tiempo del núcleo DAXPY. N deben ser parámetro de entrada al programa.

CAPTURA CÓDIGO FUENTE: daxpy.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>

#define TAM_MAX 134217728

double x[TAM_MAX];
double y[TAM_MAX];
double a=12345.6789;

int main(int argc, char ** argv){

    if((argc != 2) && (argc != 1)){
        printf("NUMERO DE PARAMETROS INTRODUCIDOS INCORRECTO\n");
        exit(EXIT_FAILURE);
    }

    int tam;
    if(argc == 2){
      tam = atoi(argv[1]);

      if(tam > TAM_MAX)
        tam = TAM_MAX;
    }
    if(argc == 1){
      tam = TAM_MAX;
    }

    for(int i=0; i<tam; i++){
        y[i] = a-i*0.1234;
        x[i] = a+i*0.1234;
    }

    struct timespec cgt1,cgt2;
    double ncgt;
```

```
38      clock_gettime(CLOCK_REALTIME,&cgt1);
39      for(int i=0; i<tam; i++){
40          y[i] = a*x[i] + y[i];s
41      }
42      clock_gettime(CLOCK_REALTIME,&cgt2);
43
44      ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+(double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));
45
46      printf("\n>> TIEMPO DE EJECUCION: %11.9f | DIMENSION: %d\n", ncgt, tam);
47
48      return EXIT_SUCCESS;
49  }
50
```

Tiempos ejec. Longitud vectores=XXXX	**-O0**	**-Os**	**-O2**	**-O3**
	0.696230389	0.32406107 1	0.32343407 7	0.31531524 1

CAPTURAS DE PANTALLA (que muestren la compilación y que el resultado es correcto)**:**

```
[e2estudiante14@atcgrid ejer3]$ srun -p ac daxpyO0

>> TIEMPO DE EJECUCION: 0.696230389 | DIMENSION: 134217728
[e2estudiante14@atcgrid ejer3]$ srun -p ac daxpyO2

>> TIEMPO DE EJECUCION: 0.323434077 | DIMENSION: 134217728
[e2estudiante14@atcgrid ejer3]$ srun -p ac daxpyO3

>> TIEMPO DE EJECUCION: 0.315315241 | DIMENSION: 134217728
[e2estudiante14@atcgrid ejer3]$ srun -p ac daxpyOs

>> TIEMPO DE EJECUCION: 0.324061071 | DIMENSION: 134217728
```

COMENTARIOS QUE EXPLIQUEN LAS DIFERENCIAS EN ENSAMBLADOR:

En cuanto a los comentarios que podemos decir, de todos los códigos de ensamblador el que mas código genera es la optimización O3, este es el que tiene el nivel mas alto de optimización pero es el que usa mas memoria que los demás niveles y genera unos binarios de tamaño mas grande, sin embargo, el código Os es el mas rápido de todos.

Por ultimo comentar que entre Os y O2 apenas se ven algunas diferencias, ya que Os se basa en la optimización del tipo O2 pero además reduce el tamaño del ejecutable.

CÓDIGO EN ENSAMBLADOR (no es necesario introducir aquí el código como captura de pantalla, ajustar el tamaño de la letra para que una instrucción no ocupe más de un renglón)**:**
(PONER AQUÍ SÓLO LA ZONA DEL CÓDIGO ENSAMBLADOR DONDE ESTÁ EL CÓDIGO EVALUADO, USE COLORES PARA DESTACAR LAS DIFERENCIAS)

daxpy00. s

```
call    clock_gettime@PLT
        movl    $0, -60(%rbp)
        jmp     .L7
.L8:
        movl    -60(%rbp), %eax
        cltq
        leaq    0(,%rax,8), %rdx
        leaq    x(%rip), %rax
        movsd   (%rdx,%rax), %xmm1
        movsd   a(%rip), %xmm0
        mulsd   %xmm0, %xmm1
        movl    -60(%rbp), %eax
        cltq
        leaq    0(,%rax,8), %rdx
        leaq    y(%rip), %rax
        movsd   (%rdx,%rax), %xmm0
        addsd   %xmm1, %xmm0
        movl    -60(%rbp), %eax
        cltq
        leaq    0(,%rax,8), %rdx
        leaq    y(%rip), %rax
        movsd   %xmm0, (%rdx,%rax)
        addl    $1, -60(%rbp)
.L7:
        movl    -60(%rbp), %eax
        cmpl    -68(%rbp), %eax
        jl      .L8
        leaq    -32(%rbp), %rax
        movq    %rax, %rsi
        movl    $0, %edi
        call
        clock_gettime@PLT
```

daxpy0s. s

```
call    clock_gettime@PLT
        movsd   a(%rip), %xmm1
        xorl    %eax, %eax
        leaq    y(%rip), %rdx
        leaq    x(%rip), %rcx
.L6:
        cmpl    %eax, %r12d
        jle     .L13
        movsd   (%rcx,%rax,8),
%xmm0
        mulsd   %xmm1, %xmm0
        addsd   (%rdx,%rax,8),
%xmm0
        movsd   %xmm0,
(%rdx,%rax,8)
        incq    %rax
        jmp     .L6
.L13:
        xorl    %edi, %edi
        leaq    24(%rsp), %rsi
        call
        clock_gettime@PLT
```

daxpy02. s

```
call    clock_gettime@PLT
        leaq    x(%rip), %rdx
        leal    -1(%r12), %ecx
        movsd   a(%rip), %xmm1
        leaq    8(%rdx), %rsi
        leaq    y(%rip), %rax
        leaq    (%rsi,%rcx,8), %rcx
        .p2align 4,,10
        .p2align 3
.L6:
        movsd   (%rdx), %xmm0
        addq    $8, %rdx
        addq    $8, %rax
        mulsd   %xmm1, %xmm0
        addsd   -8(%rax), %xmm0
        movsd   %xmm0, -8(%rax)
        cmpq    %rcx, %rdx
        jne     .L6
.L7:
        xorl    %edi, %edi
        leaq    16(%rsp), %rsi
        call
        clock_gettime@PLT
```

daxpy03. s

```
call    clock_gettime@PLT
        movsd   a(%rip), %xmm0
.L15:
        movl    %r12d, %ecx
        movapd  %xmm0, %xmm2
        leaq    x(%rip), %rdx
        shrl    %ecx
        unpcklpd %xmm2, %xmm2
        leaq    y(%rip), %rax
        salq    $4, %rcx
        addq    %rbp, %rcx
        .p2align 4,,10
        .p2align 3
.L11:
        movapd  (%rdx), %xmm1
        addq    $16, %rdx
        addq    $16, %rax
        mulpd   %xmm2, %xmm1
        addpd   -16(%rax), %xmm1
        movaps  %xmm1, -16(%rax)
        cmpq    %rcx, %rdx
        jne     .L11
        movl    %r12d, %eax
        andl    $-2, %eax
        testb   $1, %r12b
        je      .L13
.L16:
        cltq
        mulsd   0(%rbp,%rax,8), %xmm0
        addsd   (%rbx,%rax,8), %xmm0
        movsd   %xmm0, (%rbx,%rax,8)
.L13:
        xorl    %edi, %edi
        leaq    16(%rsp), %rsi
        call    clock_gettime@PLT
        pxor    %xmm0, %xmm0
        pxor    %xmm1, %xmm1
        movl    %r12d, %edx
        movq    24(%rsp), %rax
        subq    8(%rsp), %rax
        movl    $1, %edi
        leaq    .LC6(%rip), %rsi
        cvtsi2sdq %rax, %xmm0
        movq    16(%rsp), %rax
        subq    (%rsp), %rax
        divsd   .LC5(%rip), %xmm0
        cvtsi2sdq %rax, %xmm1
        movl    $1, %eax
        addsd   %xmm1, %xmm0
        call    __printf_chk@PLT
        movq    40(%rsp), %rax
        xorq    %fs:40, %rax
        jne     .L28
        addq    $48, %rsp
        .cfi_remember_state
        .cfi_def_cfa_offset 32
        xorl    %eax, %eax
        popq    %rbx
        .cfi_def_cfa_offset 24
        popq    %rbp
        .cfi_def_cfa_offset 16
        popq    %r12
        .cfi_def_cfa_offset 8
        ret
.L27:
        .cfi_restore_state
        movq    8(%rsi), %rdi
        movl    $10, %edx
        xorl    %esi, %esi
        movl    $134217728, %r12d
        call    strtol@PLT
        cmpl    $134217728, %eax
        cmovle  %eax, %r12d
        testl   %eax, %eax
        jle     .L29
        movsd   a(%rip), %xmm6
        movl    %r12d, %esi
        cmpl    $3, %eax
        jg      .L6
        xorl    %eax, %eax
        leaq    y(%rip), %rbx
        leaq    x(%rip), %rbp
        jmp     .L7
.L9:
        movq    %rsp, %rsi
        xorl    %edi, %edi
        call
        clock_gettime@PLT
```

4. **(a)** Paralizar con OpenMP en la CPU el código de la multiplicación resultante en el Ejercicio 1.(b). NOTA: usar para generar los valores aleatorios, por ejemplo, `drand48_r()`.

(b) Calcular la ganancia en prestaciones que se obtiene en atcgrid4 para el máximo número de procesadores físicos con respecto al código inicial no optimizado del Ejercicio 1.(a) para dos tamaños de la matriz.

(a) MULTIPLICACIÓN DE MATRICES PARALELO:

CAPTURA CÓDIGO FUENTE: `pmm-paralelo.c`

```c
#include <stdlib.h> // biblioteca con funciones atoi(), malloc() y free()
#include <stdio.h>  // biblioteca donde se encuentra la funciÃ³n printf()
#include <time.h>   // biblioteca donde se encuentra la funciÃ³n clock_gettime()


//#define VECTOR_LOCAL  // descomentar para que los vectores sean variables ...
//#define VECTOR_GLOBAL // descomentar para que los vectores sean variables ...
#define VECTOR_DYNAMIC  // descomentar para que los vectores sean variables ...


#ifdef VECTOR_GLOBAL
#define MAX 33554432    //=2^25
//#define MAX 4294967295  //=2^32 -1

double m1[MAX], m2[MAX], m3[MAX];
#endif

int main(int argc, char** argv){

  int i, j, k, suma;

  struct timespec cgt1,cgt2; double ncgt; //para tiempo de ejecuciÃ³n

  //Leer argumento de entrada (nÂº de componentes del vector)
  if (argc<2){
    printf("Faltan nÂº componentes del vector\n");
    exit(-1);
  }

  unsigned int N = atoi(argv[1]);
  #ifdef VECTOR_LOCAL
  double m1[N], m2[N], m3[N];   // TamaÃ±o variable local en tiempo de ejecuciÃ³n ...
                // disponible en C a partir de C99
  #endif

  #ifdef VECTOR_GLOBAL
  if (N>MAX) N=MAX;
  #endif

  // Definir las matrices
  #ifdef VECTOR_DYNAMIC

  int **m1, **m2, **m3, **tmp;
  m1 = (int**) malloc(N*sizeof(int*));
  m2 = (int**) malloc(N*sizeof(int*));
  m3 = (int**) malloc(N*sizeof(int*));
  tmp = (int**) malloc(N*sizeof(int*));

  for(i=0; i<N; i++){
      m1[i] = (int *) malloc(N*sizeof(int));
      m2[i] = (int *) malloc(N*sizeof(int));
      m3[i] = (int *) malloc(N*sizeof(int));
      tmp[i] = (int *) malloc(N*sizeof(int));
  }


   if ( m1==NULL || m2==NULL || m3==NULL || tmp==NULL ){
      printf("Error en la reserva de espacio para la matriz");
      exit(-2);

   }
  #endif

  srand(time(0));

  //Inicializar matrices
  if (N>8){

       for(i=0; i<N; i++){
           for(j=0; j<N; j++){
               m1[i][j] = rand();
               m2[i][j] = rand();
               m3[i][j] = 0;
       }
  }

  else {

      for(i=0; i<N; i++){
           for(j=0; j<N; j++){
               m1[i][j] = N+i+2;
               m2[i][j] = N+i+4;
               m3[i][j] = 0;
           }
      }

  }

  for(i=0; i<N; i++){
       for(j=0; j<N; j++){
           tmp[i][j] = m2[j][i];
       }
   }


// Metodo de optimizacion: trasponemos la matriz M2 para conseguir accesos secuenciales

clock_gettime(CLOCK_REALTIME,&cgt1);

    for (i = 0; i < N; i++){
       for (j = 0; j < N; ++j){
          #pragma omp parallel for reduction(+:suma)
          for (k = 0; k < N; ++k){
              suma += m1[i][k] * tmp[j][k];
          }

       m3[i][j] = suma;
       suma = 0;
    }
}

clock_gettime(CLOCK_REALTIME,&cgt2);

for(i=0; i<N; i++)
   for(j=0; j<N; j++)
      suma += m3[i][j];

ncgt=(double) (cgt2.tv_sec-cgt1.tv_sec)+
     (double) ((cgt2.tv_nsec-cgt1.tv_nsec)/(1.e+9));

 //Imprimir resultado de la multiplicacion y el tiempo de ejecucion

 printf("Tiempo:%11.9f\t / Tamanio Matriz:%u / Resultado Matriz:%u \n",ncgt,N, suma);


 #ifdef VECTOR_DYNAMIC

  for(i=0; i<N; i++){
     free(m1[i]);
     free(m2[i]);
     free(m3[i]);
  }

  free(m1); // libera el espacio reservado para v1
  free(m2); // libera el espacio reservado para v2
  free(m3); // libera el espacio reservado para v3

 #endif
 return 0;
}
```

(b) RESPUESTA

→ Para la matriz con 1000 elementos tenemos:

```
[e2estudiante14@atcgrid ejer4]$ srun -p ac4 -c32 pmm-paral
elo 1000
Tiempo:0.091021331       / Tamanio Matriz:1000 / Resultado
 Matriz:2286628718
[e2estudiante14@atcgrid ejer4]$ srun -p ac4 -c32 pmm-secue
ncial 1000
Tiempo:1.354324254       / Tamanio Matriz:1000 / Resultado
 Matriz:4196887
```

Por tanto la ganancia conseguida es de:

1.354324254/0.091021331 = 14.87919633

→Para la matriz con 500 elementos tenemos:

```
[e2estudiante14@atcgrid ejer4]$ srun -p ac4 -c32 pmm-paral
elo 500
Tiempo:0.039190476       / Tamanio Matriz:500 / Resultado
Matriz:2942922177
[e2estudiante14@atcgrid ejer4]$ srun -p ac4 -c32 pmm-secue
ncial 500
Tiempo:0.206044420       / Tamanio Matriz:500 / Resultado
Matriz:4196887
```

Por tanto la ganancia conseguida es de:

0.206044420/ 0.039190476 = 5.257512565