פרוייקט תוכנה - תרגיל 2#

מגישים:
אריאל סטולרמן
ודים סטוטלנד

(2.1)

multiply.c (includes implementation of mult_block for 2.2\a):

```
#include "multiply.h"

/* Question 2.1, 1(a) */

void  mult_ijk(elem A[], elem B[], elem C[], int n){
      int i,j,k;
      for (i=0; i<n; i++)
            for (j=0; j<n; j++){
                  int sum = 0;
                  for (k=0; k<n; k++){
                        sum += A[i+k*n] * B[k+j*n];
                  }
                  C[i+j*n] = sum;
            }
}

void  mult_ikj(elem A[], elem B[], elem C[], int n){
      int i,j,k;
      for (i=0; i<n; i++)
            for (k=0; k<n; k++){
                  int local = A[i+k*n];
                  for (j=0; j<n; j++)
                        C[i+j*n] += local * B[k+j*n];
            }
}

void  mult_jik(elem A[], elem B[], elem C[], int n){
      int i,j,k;
      for (j=0; j<n; j++)
            for (i=0; i<n; i++){
                  int sum = 0;
                  for (k=0; k<n; k++){
                        sum += A[i+k*n] * B[k+j*n];
                  }
                  C[i+j*n] = sum;
            }
}

void  mult_jki(elem A[], elem B[], elem C[], int n){
      int i,j,k;
      for (j=0; j<n; j++)
            for (k=0; k<n; k++){
                  int local = B[k+j*n];
                  for (i=0; i<n; i++)
                        C[i+j*n] += A[i+k*n] * local;
            }
}
```

```
void  mult_kij(elem A[], elem B[], elem C[], int n){
      int i,j,k;
      for (k=0; k<n; k++)
            for (i=0; i<n; i++){
                  int local = A[i+k*n];
                  for (j=0; j<n; j++)
                        C[i+j*n] += local * B[k+j*n];
            }
}

void  mult_kji(elem A[], elem B[], elem C[], int n){
      int i,j,k;
      for (k=0; k<n; k++)
            for (j=0; j<n; j++){
                  int local = B[k+j*n];
                  for (i=0; i<n; i++)
                        C[i+j*n] += A[i+k*n] * local;
            }
}

/* Question 2.2, 1(a) */

/* Regular matrix multiplication for r*r blocks by jki */
void mult_jki_blocks(elem A[], elem B[], elem C[], int n, int r){
        int i,j,k;
        for (j=0; j<r; j++)
                for (k=0; k<r; k++){
                        int local = B[k+j*n];
                        for (i=0; i<r; i++)
                                C[i+j*n] += A[i+k*n] * local;
                }
}

/* Blocks multiplication by jki*/
void mult_block(elem A[], elem B[], elem C[], int n, int r){
      int i,j,k;
      for (j=0; j<(n/r); j++)
            for (k=0; k<(n/r); k++)
                  for (i=0; i<(n/r); i++)
                        mult_jki_blocks(&A[i*r+k*r*n], &B[k*r+j*r*n], &C[i*r+j*r*n], n,
r);
}
```

matrix_manipulate.c:

```
#include "matrix_manipulate.h"

/* Question 2.1, 1(d) */

void  fill_matrix(elem a[], int n){
      int i;
      for (i=0; i<(n*n-1); i++)
            a[i]= (rand()%100 - 50);
}
```

mlpl.c:

```
#include "matrix.h"
#include "allocate_free.h"
#include "matrix_manipulate.h"
#include "multiply.h"

int main(void){
      int k, n, j;
      elem *A, *B, *C;
      clock_t t1, t2;
```

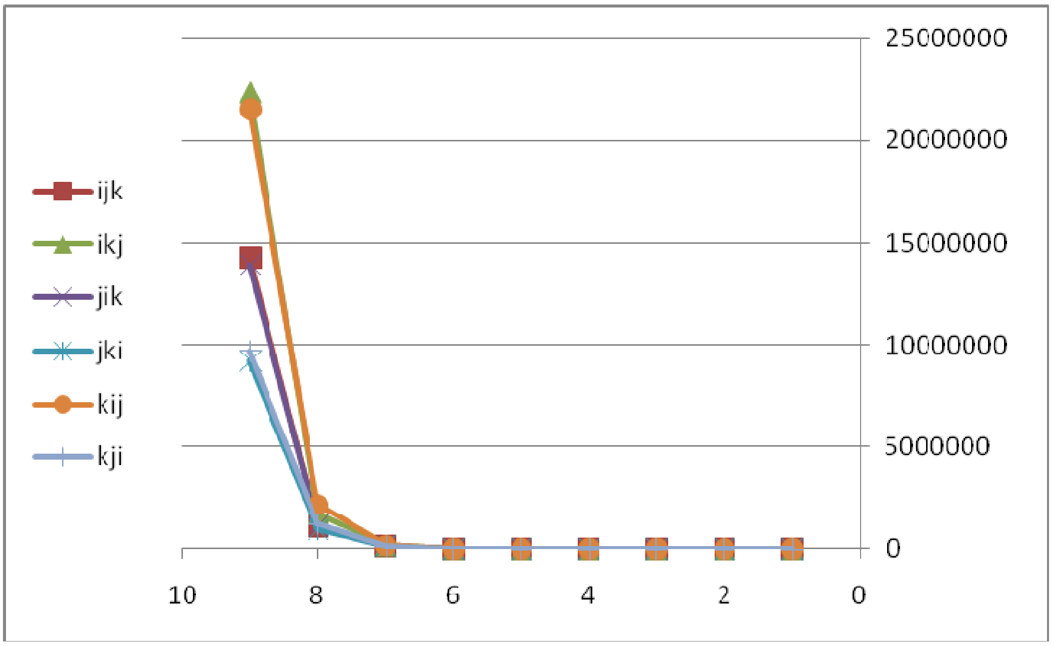
```
scanf("%d", &k);

if (k <= 0) {
      for (n=4; n<1025; n*=2){      /* runs on 4,8,16,32,64,128,256,512,1024 */

            /* Allocation space for A,B,C; random-filling and initializing */
            A=get_matrix_space(n);
            fill_matrix(A, n);
            B=get_matrix_space(n);
            fill_matrix(B, n);
            C=get_matrix_space(n);
            for (j=0; j<(n*n);j++) C[j]=0;

            printf("%d , ", n);

            /* Running all possibilities and timing */
            t1=clock();
            mult_ijk(A, B, C, n);
            t2=clock()-t1;
            printf("%ld , ", t2);
            for (j=0; j<(n*n);j++) C[j]=0;

            t1=clock();
            mult_ikj(A, B, C, n);
            t2=clock()-t1;
            printf("%ld , ", t2);
            for (j=0; j<(n*n);j++) C[j]=0;

            t1=clock();
            mult_jik(A, B, C, n);
            t2=clock()-t1;
            printf("%ld , ", t2);
            for (j=0; j<(n*n);j++) C[j]=0;

            t1=clock();
            mult_jki(A, B, C, n);
            t2=clock()-t1;
            printf("%ld , ", t2);
            for (j=0; j<(n*n);j++) C[j]=0;

            t1=clock();
            mult_kij(A, B, C, n);
            t2=clock()-t1;
            printf("%ld , ", t2);
            for (j=0; j<(n*n);j++) C[j]=0;

            t1=clock();
            mult_kji(A, B, C, n);
            t2=clock()-t1;
            printf("%ld\n", t2);
            for (j=0; j<(n*n);j++) C[j]=0;

            /* Freeing space */
            free_matrix_space(A);
            free_matrix_space(B);
            free_matrix_space(C);
      }
} else {                /* case k > 0 */

      /* Allocating space for A,B,C; Initializing C */
      A=get_matrix_space(k);
      B=get_matrix_space(k);
      C=get_matrix_space(k);
      for (n=0; n<(k*k);n++) C[n]=0;
```

```
        /* Read input */
        for(n=0; n<(k*k); n++) scanf("%d", &A[n]);
        for(n=0; n<(k*k); n++) scanf("%d", &B[n]);

        /* Calculating jki-multiplication and outputing result */
        mult_jki(A, B, C, k);
        printf("%d ", k);
        for (n=0; n<(k*k -1); n++) printf("%d ", C[n]);
        printf("%d\n", C[(k*k-1)]);

        /* Freeing space */
        free_matrix_space(A);
        free_matrix_space(B);
            free_matrix_space(C);

    }
    return 0;
}
```

2.1, (2):

להלן הנתונים שהתקבלו וגרף הנתונים:

kji	kij	jki	jik	ikj	ijk	n
0	0	0	0	0	0	4
0	0	0	0	0	0	8
0	0	0	0	0	0	16
0	0	0	0	0	0	32
0	0	10000	0	0	0	64
20000	20000	20000	20000	20000	20000	128
150000	180000	140000	140000	180000	150000	256
1220000	2210000	950000	1140000	1740000	1130000	512
9720000	21590000	9190000	13910000	22380000	14270000	1024



: gprof –b mlpl **:(3) ,2.1**

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 24.23     20.31    20.31        9     2.26     2.26  mult_ikj
 23.92     40.37    20.05        9     2.23     2.23  mult_kij
 15.49     53.35    12.99        9     1.44     1.44  mult_ijk
 15.15     66.05    12.70        9     1.41     1.41  mult_jik
 11.07     75.33     9.28        9     1.03     1.03  mult_kji
 10.27     83.94     8.61        9     0.96     0.96  mult_jki
  0.05     83.98     0.04                             main
  0.04     84.01     0.03       18     0.00     0.00  fill_matrix
  0.00     84.01     0.00       27     0.00     0.00  free_matrix_space
  0.00     84.01     0.00       27     0.00     0.00  get_matrix_space

                  Call graph

granularity: each sample hit covers 2 byte(s) for 0.01% of 84.01 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.04   83.97                 main [1]
               20.31    0.00       9/9           mult_ikj [2]
               20.05    0.00       9/9           mult_kij [3]
               12.99    0.00       9/9           mult_ijk [4]
               12.70    0.00       9/9           mult_jik [5]
                9.28    0.00       9/9           mult_kji [6]
                8.61    0.00       9/9           mult_jki [7]
                0.03    0.00      18/18          fill_matrix [8]
                0.00    0.00      27/27          get_matrix_space [10]
                0.00    0.00      27/27          free_matrix_space [9]
-----------------------------------------------
               20.31    0.00       9/9           main [1]
[2]     24.2   20.31    0.00       9         mult_ikj [2]
-----------------------------------------------
               20.05    0.00       9/9           main [1]
[3]     23.9   20.05    0.00       9         mult_kij [3]
-----------------------------------------------
               12.99    0.00       9/9           main [1]
[4]     15.5   12.99    0.00       9         mult_ijk [4]
-----------------------------------------------
               12.70    0.00       9/9           main [1]
[5]     15.1   12.70    0.00       9         mult_jik [5]
-----------------------------------------------
                9.28    0.00       9/9           main [1]
[6]     11.0    9.28    0.00       9         mult_kji [6]
-----------------------------------------------
                8.61    0.00       9/9           main [1]
[7]     10.2    8.61    0.00       9         mult_jki [7]
-----------------------------------------------
                0.03    0.00      18/18          main [1]
[8]      0.0    0.03    0.00      18         fill_matrix [8]
-----------------------------------------------
                0.00    0.00      27/27          main [1]
[9]      0.0    0.00    0.00      27         free_matrix_space [9]
-----------------------------------------------
                0.00    0.00      27/27          main [1]
[10]     0.0    0.00    0.00      27         get_matrix_space [10]
-----------------------------------------------
```

```
Index by function name

   [8] fill_matrix            [4] mult_ijk              [3] mult_kij
   [9] free_matrix_space      [2] mult_ikj              [6] mult_kji
  [10] get_matrix_space       [5] mult_jik
   [1] main                   [7] mult_jki
```

2.1, (4):

לפי ניתוח תוצאות ה-gprof, נראה כי שיטת ה-jki היא המהירה ביותר, וזאת כיוון שהזמן המצטבר של הרצתה בכל אחת מהבדיקות (4 עד 1024) היה הנמוך ביותר, כלומר בממוצע על גודל המטריצה, שיטה זו היא היעילה ביותר. תוצאה זו נובעת ככל הנראה מכך שבשיטה זו הגישה לזיכרון היא היעילה ביותר (שוב, בממוצע על גודל המטריצה). גם בגרף המתאר את זמני הריצה ניתן לראות כי jki ממוקמת יחסית נמוך, בייחוד בקלטים גבוהים בהם שיטות אחרות מקבלות ערכים גבוהים.

2.1, (5):

```
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 23
model name  : Intel(R) Xeon(R) CPU           E5410  @ 2.33GHz
stepping    : 8
cpu MHz           : 2333.412
cache size  : 6144 KB
fpu         : yes
fpu_exception     : yes
cpuid level : 10
wp          : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss syscall nx lm constant_tsc arch_perfmon pebs bts
rep_good pni ssse3 cx16 sse4_1 lahf_lm
bogomips    : 4673.73
clflush size      : 64
cache_alignment   : 64
address sizes     : 36 bits physical, 48 bits virtual
power management:

processor   : 1
vendor_id   : GenuineIntel
cpu family  : 6
model       : 23
model name  : Intel(R) Xeon(R) CPU           E5410  @ 2.33GHz
stepping    : 8
cpu MHz           : 2333.412
cache size  : 6144 KB
fpu         : yes
fpu_exception     : yes
cpuid level : 10
wp          : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss syscall nx lm constant_tsc arch_perfmon pebs bts
rep_good pni ssse3 cx16 sse4_1 lahf_lm
bogomips    : 4667.65
clflush size      : 64
cache_alignment   : 64
address sizes     : 36 bits physical, 48 bits virtual
power management:

processor   : 2
vendor_id   : GenuineIntel
cpu family  : 6
model       : 23
```

```
model name  : Intel(R) Xeon(R) CPU           E5410  @ 2.33GHz
stepping    : 8
cpu MHz           : 2333.412
cache size  : 6144 KB
fpu         : yes
fpu_exception     : yes
cpuid level : 10
wp          : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss syscall nx lm constant_tsc arch_perfmon pebs bts
rep_good pni ssse3 cx16 sse4_1 lahf_lm
bogomips    : 4667.79
clflush size      : 64
cache_alignment   : 64
address sizes     : 36 bits physical, 48 bits virtual
power management:

processor   : 3
vendor_id   : GenuineIntel
cpu family  : 6
model       : 23
model name  : Intel(R) Xeon(R) CPU           E5410  @ 2.33GHz
stepping    : 8
cpu MHz           : 2333.412
cache size  : 6144 KB
fpu         : yes
fpu_exception     : yes
cpuid level : 10
wp          : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss syscall nx lm constant_tsc arch_perfmon pebs bts
rep_good pni ssse3 cx16 sse4_1 lahf_lm
bogomips    : 4667.33
clflush size      : 64
cache_alignment   : 64
address sizes     : 36 bits physical, 48 bits virtual
power management:
```

(2.2)

block_mlpl:

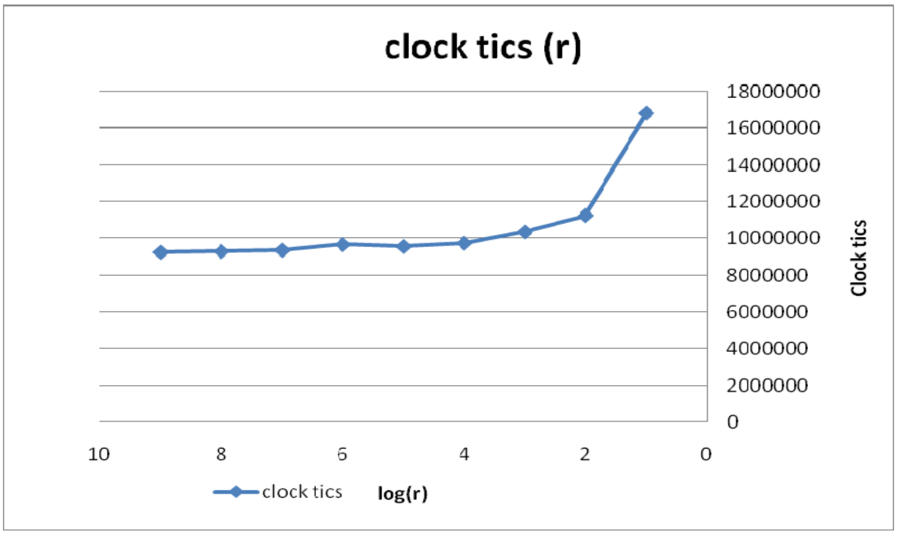
```
#include "matrix.h"
#include "allocate_free.h"
#include "matrix_manipulate.h"
#include "multiply.h"

/* implementing power function */
int intpow(int a, int b){
      int i, prod;
      prod = 1;
      for (i=0; i<b; i++) prod *= a;
      return prod;
}

int main(void){
      int k, n, j, l;
      elem *A, *B, *C;
      clock_t t1, t2;
      scanf("%d", &k);

      if (k <= 0) {                 /* case k <= 0 */
            for (n=4; n<513; n*=2){

                  /* Allocatiing space for A,B,C; Fill with random values, initializing
*/
                  A=get_matrix_space(n);
                  fill_matrix(A, n);
                  B=get_matrix_space(n);
                  fill_matrix(B, n);
                  C=get_matrix_space(n);
                  for (j=0; j<(n*n);j++) C[j]=0;

                  /* Printing n, r=n/2 and mult_block run-time */
                  printf("%d , %d , ", n, n/2);
                  t1=clock();
                  mult_block(A, B, C, n, n/2);
                  t2=clock()-t1;
                  printf("%ld\n", t2);

                  /* Freeing space */
                  free_matrix_space(A);
                  free_matrix_space(B);
                  free_matrix_space(C);
            }

            /* Allocating space for case 1024 matrices, filling and initializing */
            A=get_matrix_space(1024);
            B=get_matrix_space(1024);
            C=get_matrix_space(1024);
            fill_matrix(A, 1024);
            fill_matrix(B, 1024);

            /* Running mult_block with r=2^1...2^10, printing results */
            for (l=1; l<10; l++){
                  for (j=0; j<(1024*1024); j++) C[j]=0;
                  t1=clock();
                        mult_block(A, B, C, n, intpow(2, l));
                        t2=clock()-t1;
                        printf("1024 , %d , %ld\n", intpow(2, l), t2);
            }
```

```
    } else {                /* case k > 0 */

        /* Allocating space, get inputs for A, B, initializing C */
        A=get_matrix_space(k);
        B=get_matrix_space(k);
        C=get_matrix_space(k);
        for (n=0; n<(k*k);n++) C[n]=0;
        for(n=0; n<(k*k); n++) scanf("%d", &A[n]);
        for(n=0; n<(k*k); n++) scanf("%d", &B[n]);

        /* Run mult_block with r=min(k/2, 512) */
        if (k/2 < 512) mult_block(A, B, C, k, k/2);
        else mult_block(A, B, C, k, 512);

        /* Printing C as in case k > 0 in mlpl */
        printf("%d ", k);
        for (n=0; n<(k*k -1); n++) printf("%d ", C[n]);
        printf("%d\n", C[(k*k-1)]);

        /* Freeing memory */
        free_matrix_space(A);
        free_matrix_space(B);
        free_matrix_space(C);

    }
    return 0;
}
```

2.2, (2):

להלן הנתונים שהתקבלו וגרף הנתונים:

r	log(r)	clock tics
2	1	16810000
4	2	11260000
8	3	10370000
16	4	9750000
32	5	9570000
64	6	9670000
128	7	9370000
256	8	9300000
512	9	9260000



2.2, (3):

מניתוח התוצאות נראה כי בלוק בגודל 512 נותן את התוצאה הטובה ביותר, וזאת כנראה מכיוון שהגישה לזיכרון בשיטה זו היא היעילה ביותר – כלומר חישוב בלוקים בגודל 512 מאפשרת (מכל גדלי הבלוקים שנבדקו בשאלה) את הגישה היעילה ביותר בזיכרון. ניתן לראות זאת באופן ברור בגרף.