

LAB3. Long Latency Operations

Primer timing:

```
16:17 quim: ~/UPC/PCA/PCA-FIB/LAB3/lab3_session/primers [main]$ ../../scripts/autopca -e ./primers.0
[i]    Accounting de ./primers.0, numero de repeticions: 1

      Max. elapsed:   .83 seconds
      Min. elapsed:   .83 seconds
      Avg. elapsed:   .8300 seconds

      Max. CPU time:  .81 seconds
      Min. CPU time:  .81 seconds
      Avg. CPU time:  .8100 seconds

      Max. CPU:       97%
      Min. CPU:       97%
      Avg. CPU:       97.00%
```

Primer profiling amb gprof. Veiem una latència important a les línies 47, 48, 56 i 57 que pertanyen a les funcions *clearBit* i *getBit*:

```
Each sample counts as 0.01 seconds.
%   cumulative   self           calls     self        total    name
time seconds    seconds             ns/call     ns/call                ns/call
22.86   0.10   0.10                  23492030    0.43    0.43  clearBit (primers.c:47 @ 400910)
18.29   0.18   0.08                  10003162    1.01    1.01  getBit (primers.c:56 @ 40098d)
16.00   0.25   0.07                                0.00    0.00  clearBit (primers.c:48 @ 400936)
13.71   0.31   0.06                                0.00    0.00  getBit (primers.c:57 @ 4009b3)
6.86    0.34   0.03                                0.00    0.00  clearBit (primers.c:51 @ 400964)
5.71    0.37   0.03                                0.00    0.00  clearBit (primers.c:49 @ 400950)
4.57    0.39   0.02                                0.00    0.00  getBit (primers.c:59 @ 4009d6)
2.29    0.40   0.01                                0.00    0.00  clearBit (primers.c:46 @ 4008fb)
2.29    0.41   0.01                                0.00    0.00  getBit (primers.c:55 @ 400978)
2.29    0.42   0.01                                0.00    0.00  clearBit (primers.c:50 @ 400961)
2.29    0.43   0.01                                0.00    0.00  getBit (primers.c:58 @ 4009cd)
1.14    0.43   0.01                                0.00    0.00  clearBit (primers.c:52 @ 400975)
1.14    0.44   0.01                                0.00    0.00  getBit (primers.c:60 @ 4009e7)
1.14    0.44   0.01                                0.00    0.00  getBit (primers.c:61 @ 4009f1)
0.00    0.44   0.00    664579          0.00    0.00  printPrime (primers.c:80 @ 400a85)
0.00    0.44   0.00           1          0.00    0.00  createBitArray (primers.c:24 @ 40078b)
0.00    0.44   0.00           1          0.00    0.00  findPrimes (primers.c:85 @ 400aaf)
0.00    0.44   0.00           1          0.00    0.00  freeBitArray (primers.c:18 @ 400777)
0.00    0.44   0.00           1          0.00    0.00  setAll (primers.c:72 @ 400a3c)
```

Call graph

Mirant el codi veiem que estem dividint i fent el mòdul amb el nombre 32 que és potència de 2. Podem usar BitHacks per a canviar aquesta operació per una de menys costosa.

Codi Original:

```

    ba->bytesPerInt = sizeof(unsigned int);
    ba->bitsPerInt = ba->bitsPerByte * ba->bytesPerInt;
    ba->bytesPerInt = sizeof(unsigned int);
    ba->bitsInArray = bits;
    ba->intsInArray = bits / ba->bitsPerInt + 1;
    ba->p = malloc(ba->intsInArray * ba->bytesPerInt);
    assert(ba->p != NULL);
    return ba;
}

void setBit(BITARRAY * ba, bignum bitSS)
{
    unsigned int *pInt = ba->p + (bitSS / ba->bitsPerInt);
    unsigned int remainder = (bitSS % ba->bitsPerInt);
    *pInt |= (1 << remainder);
}

void clearBit(BITARRAY * ba, bignum bitSS)
{
    unsigned int *pInt = ba->p + (bitSS / ba->bitsPerInt);
    unsigned int remainder = (bitSS % ba->bitsPerInt);
    unsigned int mask = 1 << remainder;
    mask = ~mask;
    *pInt &= mask;
}

int getBit(BITARRAY * ba, bignum bitSS)
{
    unsigned int *pInt = ba->p + (bitSS / ba->bitsPerInt);
    unsigned int remainder = (bitSS % ba->bitsPerInt);
    unsigned int ret = *pInt;
    ret &= (1 << remainder);
    return (ret != 0);
}

```

Optimitzacions:

```
46
45 void setBit(BITARRAY * ba, bignum bitSS)
44 {
43     unsigned int *pInt = ba->p + (bitSS >> 5);
42     unsigned int remainder = (bitSS & 31);
41     *pInt |= (1 << remainder);
40 }
39
38 void clearBit(BITARRAY * ba, bignum bitSS)
37 {
36     unsigned int *pInt = ba->p + (bitSS >> 5); //Cal optimitzar: unsigned int *pInt = ba->p + (bitSS / ba->bitsPerInt)
35     unsigned int remainder = (bitSS & 31); //Cal optimitzar
34     unsigned int mask = 1 << remainder;
33     mask = ~mask;
32     *pInt &= mask;
31 }
30
29 int getBit(BITARRAY * ba, bignum bitSS)
28 {
27     unsigned int *pInt = ba->p + (bitSS >> 5); //Cal optimitzar
26     unsigned int remainder = (bitSS & 31); //Cal optimitzar
25     unsigned int ret = *pInt;
24     ret &= (1 << remainder);
23     return (ret != 0);
22 }
```

Tornant a fer profiling veiem que les línies han canviat. Ara el màxim pes de l'execució recau en operacions de shift de bits:

```
Each sample counts as 0.01 seconds.
%   cumulative    self       self    total
time  seconds    seconds   calls  ns/call  ns/call  name
18.29    0.04    0.04
18.29    0.08    0.04
11.43    0.11    0.03
6.86     0.12    0.02
6.86     0.14    0.02
6.86     0.15    0.02
4.57     0.16    0.01 23492030    0.43    0.43 clearBit (primers.opt.c:49 @ 400934)
4.57     0.17    0.01 10003162    1.01    1.01 getBit (primers.opt.c:56 @ 400971)
4.57     0.18    0.01 clearBit (primers.opt.c:51 @ 400948)
4.57     0.19    0.01 clearBit (primers.opt.c:50 @ 400945)
4.57     0.20    0.01 findPrimes (primers.opt.c:98 @ 400b23)
2.29     0.21    0.01 findPrimes (primers.opt.c:96 @ 400b2b)
2.29     0.21    0.01 clearBit (primers.opt.c:46 @ 4008fb)
2.29     0.22    0.01 getBit (primers.opt.c:55 @ 40095c)
2.29     0.22    0.01 clearBit (primers.opt.c:47 @ 400910)
0.00     0.22    0.00 664579     0.00    0.00 findPrimes (primers.opt.c:97 @ 400b10)
0.00     0.22    0.00 1         0.00    0.00 getBit (primers.opt.c:58 @ 400995)
0.00     0.22    0.00 1         0.00    0.00 clearBit (primers.opt.c:48 @ 40092a)
0.00     0.22    0.00 1         0.00    0.00 clearBit (primers.opt.c:52 @ 400959)
0.00     0.22    0.00 1         0.00    0.00 findPrimes (primers.opt.c:100 @ 400b35)
0.00     0.22    0.00 1         0.00    0.00 findPrimes (primers.opt.c:101 @ 400b42)
0.00     0.22    0.00 1         0.00    0.00 printPrime (primers.opt.c:80 @ 400a4d)
0.00     0.22    0.00 1         0.00    0.00 createBitArray (primers.opt.c:24 @ 40078b)
0.00     0.22    0.00 1         0.00    0.00 findPrimes (primers.opt.c:85 @ 400a77)
0.00     0.22    0.00 1         0.00    0.00 freeBitArray (primers.opt.c:18 @ 400777)
0.00     0.22    0.00 1         0.00    0.00 setAll (primers.opt.c:72 @ 400a04)
```

I l'speedup que aconseguim és de 2.8 i creiem que ens podem considerar satisfets.

```

PCA/PCA-FIB> scripts/autopca -e LAB3/lab3_session/primers/primers.opt.0 -g LAB3/lab3_session/primers/primers.0 -n 4
[i] Comparant els outputs dels executables...
[i] Accounting de LAB3/lab3_session/primers/primers.0, numero de repeticions: 4

    Max. elapsed: .64 seconds
    Min. elapsed: .64 seconds
    Avg. elapsed: .6400 seconds

    Max. CPU time: .64 seconds
    Min. CPU time: .63 seconds
    Avg. CPU time: .6375 seconds

    Max. CPU: 99%
    Min. CPU: 99%
    Avg. CPU: 99.00%

[i] Accounting de LAB3/lab3_session/primers/primers.opt.0, numero de repeticions: 4

    Max. elapsed: .23 seconds
    Min. elapsed: .23 seconds
    Avg. elapsed: .2300 seconds

    Max. CPU time: .23 seconds
    Min. CPU time: .22 seconds
    Avg. CPU time: .2275 seconds

    Max. CPU: 99%
    Min. CPU: 99%
    Avg. CPU: 99.00%

[i] Calcul del Speedup
    Speedup elapsed: 2.7826
    Speedup CPU: 2.8021

```

Si compilem la nostra versió de primers amb -O2 i -O3 obtenim un speedup de 8.07 i 8.56 respectivament.

Speedup amb O2:

```

16:39 quim: ~/UPC/PCA/PCA-FIB/LAB3/lab3_session/primers [main]$ ../../scripts/autopca -e ./primers.opt.2 -g ./primers.0 -n 4
[i] Comparant els outputs dels executables...
[i] Accounting de ./primers.0, numero de repeticions: 4

    Max. elapsed: .79 seconds
    Min. elapsed: .78 seconds
    Avg. elapsed: .7875 seconds

    Max. CPU time: .79 seconds
    Min. CPU time: .77 seconds
    Avg. CPU time: .7800 seconds

    Max. CPU: 99%
    Min. CPU: 99%
    Avg. CPU: 99.00%

[i] Accounting de ./primers.opt.2, numero de repeticions: 4

    Max. elapsed: .10 seconds
    Min. elapsed: .09 seconds
    Avg. elapsed: .0975 seconds

    Max. CPU time: .10 seconds
    Min. CPU time: .09 seconds
    Avg. CPU time: .0950 seconds

    Max. CPU: 100%
    Min. CPU: 99%
    Avg. CPU: 99.50%

[i] Calcul del Speedup
    Speedup elapsed: 8.0769
    Speedup CPU: 8.2105
16:39 quim: ~/UPC/PCA/PCA-FIB/LAB3/lab3_session/primers [main]$

```

Speedup amb O3:

```

16:39 quim: ~/UPC/PCA/PCA-FIB/LAB3/lab3_session/primers [main]$ ../../scripts/autopca -e ./primers.opt.3 -g ./primers.0
[i] Comparant els outputs dels executables...
[i] Accounting de ./primers.0, numero de repeticions: 4

    Max. elapsed: .80 seconds
    Min. elapsed: .79 seconds
    Avg. elapsed: .7925 seconds

    Max. CPU time: .80 seconds
    Min. CPU time: .78 seconds
    Avg. CPU time: .7875 seconds

    Max. CPU: 99%
    Min. CPU: 99%
    Avg. CPU: 99.00%

[i] Accounting de ./primers.opt.3, numero de repeticions: 4

    Max. elapsed: .10 seconds
    Min. elapsed: .09 seconds
    Avg. elapsed: .0925 seconds

    Max. CPU time: .09 seconds
    Min. CPU time: .09 seconds
    Avg. CPU time: .0900 seconds

    Max. CPU: 100%
    Min. CPU: 98%
    Avg. CPU: 98.75%

[i] Calcul del Speedup
    Speedup elapsed: 8.5675
    Speedup CPU: 8.7500
16:40 quim: ~/UPC/PCA/PCA-FIB/LAB3/lab3_session/primers [main]$

```

Amb el output d'insmix podem indagar en aquest comportament veient que s'està reduint considerablement el nombre de divisions entre el programa compilat amb 02 (dreta) i 03 (esquerra).

#	opcode	count-unpredicated	count-predicated	#	opcode	count-unpredicated	count-predicated
1	8 ADD	53359321	0	51	8 ADD	57000944	0
0	31 AND	60979631	0	50	31 AND	30142794	0
9	63 BSF	193	0	49	63 BSF	189	0
8	64 BSR	3	0	48	64 BSR	3	0
7	66 BT	123	0	47	66 BT	122	0
6	75 CALL_NEAR	9304874	0	46	75 CALL_NEAR	9304864	0
5	78 CDQE	3322898	0	45	78 CDQE	3322898	0
4	92 CMOVB	40	0	44	92 CMOVB	40	0
3	93 CMOVBE	1329158	0	43	93 CMOVBE	1329158	0
2	97 CMOVNBE	6	0	42	97 CMOVNBE	6	0
1	103 CMOVNZ	664598	0	41	103 CMOVNZ	664597	0
0	106 CMOVS	664579	0	40	106 CMOVS	664579	0
9	107 CMOVZ	120	0	39	107 CMOVZ	119	0
8	108 CMP	83182426	0	38	108 CMP	83494909	0
7	117 CMPXCHG	664593	0	37	117 CMPXCHG	664593	0
6	122 CMPXCHG_LOCK	2	0	36	122 CMPXCHG_LOCK	2	0
5	125 CPUID	27	0	35	125 CPUID	27	0
4	126 CQO	2	0	34	126 CQO	2	0
3	154 DEC	664593	0	33	154 DEC	664593	0
2	156 DIV	664673	0	32	156 DIV	34159863	0
1	283 IDIV	2	0	31	283 IDIV	2	0
0	284 IMUL	478	0	30	284 IMUL	478	0
9	286 INC	1	0	29	286 INC	1	0
8	309 JB	1996646	0	28	309 JB	1996645	0
7	310 JBE	3989380	0	27	310 JBE	3989377	0
6	313 JL	1329161	0	26	313 JL	1329161	0
5	314 JLE	1993847	0	25	314 JLE	1993846	0
4	315 JMP	16619279	0	24	315 JMP	16619285	0
3	317 JNB	28809925	0	23	317 JNB	28809924	0
2	318 JNBE	6647610	0	22	318 JNBE	6647616	0

Programa pi.c

El programa que expliquem en aquesta secció és el que hem penjat al servidor com a *pi.opt3.c*. Hem aplicat les següents optimitzacions al codi:

- Hem creat funcions especialitzades per a fer la divisió de 5, 25 i 239.

- Hem aplicat la tècnica de *memoization* per a pre-calcular els valors que poden donar el quocient i el residu en cada una de les iteracions de la funció que divideix entre aquests nombres.

Per a fer-ho, hem creat 2 vectors per a cada especialització que emmagatzemen els valors possibles de q i r:

```
int memo_q5[50];  
int memo_r5[50];  
  
int memo_q25[250];  
int memo_r25[250];  
  
int memo_q239[2390];  
int memo_r239[2390];
```

Aquests vector s'inicialitzen a l'inci del programa amb una funció d'inicialització que accepta com a paràmetres els vectors i el divisor:

```
void init_memo(int *q, int *r, int num){  
    int i,j,div;  
    i=0;  
    div=0;  
    while (i<num*10){  
        for (j=0; j<num; j++){  
            q[i] = div;  
            r[i] = j;  
            i++;  
        }  
        div++;  
    }  
}
```

I posteriorment procedim a l'especialització de la funció DIVIDE:

```
void DIVIDE_239( signed char *x)
{
    int j, k;
    unsigned q, r, u;
    long v;

    r = 0;
    for( k = 0; k <= N4; k++ )
    {
        u = r * 10 + x[k];
        x[k] = memo_q239[u];
        r = memo_r239[u];
        //x[k] = q; //resultat

    }
}
```

```
void DIVIDE_25( signed char *x)
{
    int j, k;
    unsigned q, r, u;
    long v;

    r = 0;
    for( k = 0; k <= N4; k++ )
    {
        u = r * 10 + x[k];
        x[k] = memo_q25[u];
        r = memo_r25[u];
        //x[k] = q; //resultat

    }
}
```

```

void DIVIDE_5( signed char *x)
{
    int j, k;
    unsigned q, r, u;
    long v;

    r = 0;
    for( k = 0; k <= N4; k++ )
    {
        u = r * 10 + x[k];
        x[k] = memo_q5[u];
        r = memo_r5[u];
        //x[k] = q; //resultat
    }
}

```

Aconseguiu obtenir un speed-up de 1.4 respecte l'original quan compliem amb O0:

```

dhap0@kali:~/UNI/pca/PCA-FIB/LAB3/lab3_session/pi$ ../../scripts/autopca -e ./pi.opt.0 -g ./pi.0 -n 10
[i] Comparant els outputs dels executables...
[i] Accounting de ./pi.0, numero de repeticions: 10

Max. elapsed: 10.61 seconds
Min. elapsed: 10.57 seconds
Avg. elapsed: 10.5830 seconds

Max. CPU time: 10.61 seconds
Min. CPU time: 10.56 seconds
Avg. CPU time: 10.5770 seconds

Max. CPU: 100%
Min. CPU: 99%
Avg. CPU: 99.20%

[i] Accounting de ./pi.opt.0, numero de repeticions: 10

Max. elapsed: 7.44 seconds (use "git add" and/or "git commit -a")
Min. elapsed: 7.34 seconds (LAB3/lab3_session/trigon1 vim trigon.opt1.c
Avg. elapsed: 7.3650 seconds (LAB3/lab3_session/trigon1 vim trigon.opt2.c
Max. CPU time: 7.40 seconds (opt2.c vim -static -o trigon.opt2
Min. CPU time: 7.33 seconds (c -pg -e vim -static -o trigon.opt2.pg
Avg. CPU time: 7.3500 seconds (c -pg -e vim -static -o trigon.opt2.3
Max. CPU: 100% (trigon.opt2.c -pg -e vim -static -o trigon.opt2.0
Min. CPU: 99% (trigon.opt2.c -pg -e vim -static -o trigon.opt2.g3
Avg. CPU: 99.10% (trigon.opt2.c -pg -e vim -static -o trigon.opt2.pg1
Speedup elapsed: 1.4369 (LAB3/lab3_session/trigon1 ../../scripts/autopca -e ./trigon.opt2.0 -g ./tr
Speedup CPU: 1.4390 (LAB3/lab3_session/trigon1 ../../scripts/autopca -e ./trigon.opt2.0 -g ./tr

```

Trigon.c

trigon.c és un codi calcula els valors x i y a partir dels cosinus i sinus per un angle en radians que va incrementant. Tenim dues constants PUNTS i N. La primera ens dona la quantitat de valors en la que es dividirà la circumferència i per tant el nombre de sinus i cosinus que farem per a cada repetició. N és el nombre de repeticions, és a dir, el nombre

de cops que calcularem *PUNTS* sinus i cosinus. Cada cop que es calcula un sinus i un cosinus es mostren per pantalla amb la funció *fwrite*.

Fent un profiling amb *gprof* de *trigon.c* compilat amb *-O3* obtenim els següents resultats:

```
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self           self       total
time  seconds  seconds   calls   Ts/call  Ts/call  name
47.06    0.16    0.16           1      0.00    0.00  sincosf32x
32.35    0.27    0.11           1      0.00    0.00  fwrite
14.71    0.32    0.05           1      0.00    0.00  _IO_new_file_xsputn
 5.88    0.34    0.02           1      0.00    0.00  write
 0.00    0.34    0.00           1      0.00    0.00  main (trigon.c:12 @ 401670)
```

Podem veure clarament que la crida a sistema *write* està pesant molt en l'execució del programa i també les crides a funcions trigonomètriques. Si fem el *gprof* de *trigon.c* compilat amb llibreries estàtiques el *write* triga molt menys ja que no ha de calcular l'*offset* per accedir a la *libc* de manera dinàmica:

```
Flat profile:

Each sample counts as 0.01 seconds.
 %   cumulative   self           self       total
time  seconds  seconds   calls   Ts/call  Ts/call  name
52.78    0.19    0.19           1      0.00    0.00  sincosf32x
16.67    0.25    0.06           1      0.00    0.00  _IO_new_file_xsputn
16.67    0.31    0.06           1      0.00    0.00  write
11.11    0.35    0.04           1      0.00    0.00  fwrite
 2.78    0.36    0.01           1      0.00    0.00  __memcpy_avx_unaligned_erms
 0.00    0.36    0.00           1      0.00    0.00  main (trigon.c:12 @ 401670)
```

A continuació mostrem l'*elapsed time* i el nombre de crides a sistema obtingudes amb *strace*:

```
dhap0@kali:~/UNI/pca/PCA-FIB/LAB3/lab3_session/trigon$ ../scripts/autopca -e ./trigon.0 -n 10
[i] Accounting de ./trigon.0, numero de repeticions: 10

Max. elapsed: .61 seconds
Min. elapsed: .53 seconds
Avg. elapsed: .5530 seconds

Max. CPU time: .60 seconds
Min. CPU time: .53 seconds
Avg. CPU time: .5500 seconds

Max. CPU: 100%
Min. CPU: 99%
Avg. CPU: 99.30%

dhap0@kali:~/UNI/pca/PCA-FIB/LAB3/lab3_session/trigon$ strace -c ./trigon.0 > /tmp/shit
% time    seconds  usecs/call   calls   errors syscall
-----
100.00    0.044521      1      23438          write
 0.00    0.000000      0           1          fstat
 0.00    0.000000      0           4          brk
 0.00    0.000000      0           1          execve
 0.00    0.000000      0           1          uname
 0.00    0.000000      0           1          readlink
 0.00    0.000000      0           1          arch_prctl
100.00    0.044521      1      23447          total
```

Per a millorar el problema amb el nombre de *writes* podem fer *buffering* i realitzar aquestes crides ajuntant buffers de tamany *PUNTS*. A continuació veiem el codi, l'*elapsed* i l'*strace*:

```

1 #include <math.h>
2 #include <stdio.h>
3 #include <unistd.h>, 336 MiB) copied, 134.675 s,
4 #include <string.h> sudo dd if= of=/dev/sdb statu
5 #include <stdlib.h>
6 nets.pdf
7 #define N 6000
8 #define PUNTS 1000
9 ran-10.8.0-amd64-netinst.iso
10 int main(int argc, char *argv[])
11 {rd-0.0.14.deb
12     unsigned int i, r, j, n, a;
13     double d; rant.pdf
14     double memo_cos[PUNTS], memo_sin[PUNTS];
15     double xy[PUNTS*2];
16     if (argc == 1) n = N; else n = atoi(argv[1]);
17     // $ cd ~/Downloads$ sudo dd if=openSUSE-Tumblewee
18     // password for dhap0:
19     srand(0); (163 MB, 156 MiB) copied, 50 s, 3.3 M
20     for (i=0; i<n; i++)
21     { records out
22     { r = rand(); 8 MB, 160 MiB) copied, 55.1066 s,
23     { for (j=0, d=0, a=0; j<PUNTS; j++, a+=2) // e
24     { // sword for dhap0:
25     { xy[a] = r*cos(d);
26     { xy[a+1] = r*sin(d);
27     { d += 2*M_PI/PUNTS;
28     { } remote gateway has allocated a VPN.
29     { fwrite(xy, sizeof(double), 2*PUNTS, stdout);
30     { } // pp0v -> /dev/pts/0
31     return 0; presses: [10.100.0.109], ns [84.88.52.3
32 } Negotiation complete.

```

```

dhap0@kali:~/UNI/pca/PCA-FIB/LAB3/lab3_session/trigon$ ../../scripts/autopca -e ./trigon.opt2.0 -n 10
[i] Accounting de ./trigon.opt2.0, numero de repeticions: 10

Max. elapsed: .33 seconds
Min. elapsed: .30 seconds
Avg. elapsed: .3080 seconds

Max. CPU time: .32 seconds
Min. CPU time: .29 seconds
Avg. CPU time: .3030 seconds

Max. CPU: 100%
Min. CPU: 99%
Avg. CPU: 99.20%

dhap0@kali:~/UNI/pca/PCA-FIB/LAB3/lab3_session/trigon$ strace -c ./trigon.opt2.0 > /tmp/shit
% time seconds usecs/call calls errors syscall
-----
100.00 0.038464 3 12000 0 write
0.00 0.000000 0 1 0 fstat
0.00 0.000000 0 1 0 mmap
0.00 0.000000 0 4 0 brk
0.00 0.000000 0 1 0 execve
0.00 0.000000 0 1 0 uname
0.00 0.000000 0 1 0 readlink
0.00 0.000000 0 1 0 arch_prctl
-----
100.00 0.038464 3 12010 0 total

```

I l'speedup obtingut:

```

dhap0@kali:~/UNI/pca/PCA-FIB/LAB3/lab3_session/trigon$ ../../scripts/autopca -e ./trigon.opt2.0 -g ./trigon.0 -n 10
[i] Comparant els outputs dels executables ...
[i] Accounting de ./trigon.0, numero de repeticions: 10

Max. elapsed: .60 seconds
Min. elapsed: .54 seconds
Avg. elapsed: .5650 seconds

Max. CPU time: .59 seconds
Min. CPU time: .54 seconds
Avg. CPU time: .5600 seconds

Max. CPU: 100%
Min. CPU: 99%
Avg. CPU: 99.20%

[i] Accounting de ./trigon.opt2.0, numero de repeticions: 10

Max. elapsed: .33 seconds
Min. elapsed: .30 seconds
Avg. elapsed: .3070 seconds

Max. CPU time: .33 seconds
Min. CPU time: .29 seconds
Avg. CPU time: .3020 seconds

Max. CPU: 100%
Min. CPU: 99%
Avg. CPU: 99.40%

[i] Calcul del Speedup
Speedup elapsed: 1.8403
Speedup CPU: 1.8543

```

Tornant a fer profiling podem veure com les funcions trigonomètriques ocupen el 45.35% del temps d'execució. Segons la llei d'Amdahl podem aconseguir fins a un speedup teòric màxim de 1.83 respecte la versió anterior.

Overhead	Samples	Command	Shared Object	Symbol
28.64%	346	trigon.opt2.0	trigon.opt2.0	[.] main
23.21%	284	trigon.opt2.0	trigon.opt2.0	[.] __sin_fma
22.13%	272	trigon.opt2.0	trigon.opt2.0	[.] __cos_fma
3.43%	40	trigon.opt2.0	[kernel.kallsyms]	[k] copy_user_enhanced_fast_string
1.54%	19	trigon.opt2.0	trigon.opt2.0	[.] _init
1.54%	19	trigon.opt2.0	[kernel.kallsyms]	[k] do_syscall_64
1.13%	14	trigon.opt2.0	[crc32c_intel]	[k] crc32c_pcl_intel_update
1.10%	13	trigon.opt2.0	[ext4]	[k] ext4_mark_inode_dirty

Procedim doncs a canviar el codi per a eliminar la repetició de crides a la funció sinus i cosinus. S'ha precalculat tots els valors que prendran els sinus i cosinus. Aquest codi pertany al fitxer *trigon.opt4.c* penjat al servidor de PCA:

```

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

#define N 6000
#define PUNTS 1000

int main(int argc, char *argv[])
{
    unsigned int i, r, j, n, a;
    double d;
    double memo_cos[PUNTS], memo_sin[PUNTS];
    double xy[PUNTS*2];
    if (argc == 1) n = N; else n = atoi(argv[1]);

    for(i=0, d=0; i<PUNTS; i++){
        memo_cos[i] = cos(d);
        memo_sin[i] = sin(d);
        d += 2*M_PI/PUNTS;
    }

    srand(0);
    for (i=0; i<n; i++)
    {
        r = rand();
        for (j=0, a=0; j<PUNTS; j++, a+=2)
        {
            xy[a] = r*memo_cos[j];
            xy[a+1] = r*memo_sin[j];
        }
        fwrite(xy, sizeof(double), 2*PUNTS, stdout);
    }
    return 0;
}

```

Podem doncs comparar les dues versions amb l'script que vàrem fer per al laboratori anterior per veure que efectivament l'output és idèntic i obtenim un speedup de 3.46 respecte l'original quan compilem amb O3.

```

dhap0@kali:~/UNI/pca/PCA-FIB/LAB3/lab3_session/trigon$ ../../scripts/autopca -e ./trigon.opt2.0 -g ./trigon.0 -n 10
[i] Comparant els outputs dels executables ...
[i] Accounting de ./trigon.0, numero de repeticions: 10
Max. elapsed: .67 seconds
Min. elapsed: .54 seconds
Avg. elapsed: .5680 seconds

Max. CPU time: .66 seconds
Min. CPU time: .53 seconds
Avg. CPU time: .5620 seconds

Max. CPU: 100%
Min. CPU: 99%
Avg. CPU: 99.10%

[i] Accounting de ./trigon.opt2.0, numero de repeticions: 10
Max. elapsed: .19 seconds
Min. elapsed: .16 seconds
Avg. elapsed: .1640 seconds

Max. CPU time: .19 seconds
Min. CPU time: .15 seconds
Avg. CPU time: .1580 seconds

Max. CPU: 100%
Min. CPU: 99%
Avg. CPU: 99.30%

[i] Calcul del Speedup
Speedup elapsed: 3.4634
Speedup CPU: 3.5569

```

I comparant els profilings veiem que hem aconseguit reduir l'efecte que tenia la realització de múltiples crides a les funcions *sin* i *cos*. Tanmateix, el major pes ara recau en la funció *main* degut a la inicialització dinàmica d'aquests vectors que emmagatzemen els resultats de sinus i cosinus.

perf record actual:

Samples: 668 of event 'cycles', Event count (approx.): 384704675					
Overhead	Samples	Command	Shared Object	Symbol	
59.00%	391	trigon.opt2.g0	trigon.opt2.g0	[.] main	
4.52%	29	trigon.opt2.g0	[kernel.kallsyms]	[k] copy_user_enhanced_fast_string	
2.81%	19	trigon.opt2.g0	[crc32c_intel]	[k] crc32c_pcl_intel_update	
2.69%	18	trigon.opt2.g0	[kernel.kallsyms]	[k] do_syscall_64	
1.19%	8	trigon.opt2.g0	[ext4]	[k] ext4_da_get_block_prep	

perf record del programa original:

Samples: 2K of event 'cycles', Event count (approx.): 1252793465					
Overhead	Samples	Command	Shared Object	Symbol	
21.29%	465	trigon.g0	trigon.g0	[.] __cos_fma	
17.76%	390	trigon.g0	trigon.g0	[.] _IO_fwrite	
16.61%	362	trigon.g0	trigon.g0	[.] main	
12.28%	266	trigon.g0	trigon.g0	[.] __sin_fma	
11.86%	258	trigon.g0	trigon.g0	[.] _IO_new_file_xsputn	
3.13%	68	trigon.g0	trigon.g0	[.] __memmove_avx_unaligned_erms	
1.41%	31	trigon.g0	trigon.g0	[.] _init	
1.14%	25	trigon.g0	[kernel.kallsyms]	[k] do_syscall_64	
0.86%	19	trigon.g0	[crc32c_intel]	[k] crc32c_pcl_intel_update	
0.77%	17	trigon.g0	[kernel.kallsyms]	[k] copy_user_enhanced_fast_string	