LABORATORI PCA

Projecte Final: Optimització ftdock

Felip Moll Marquès

PCA – FiB (UPC)

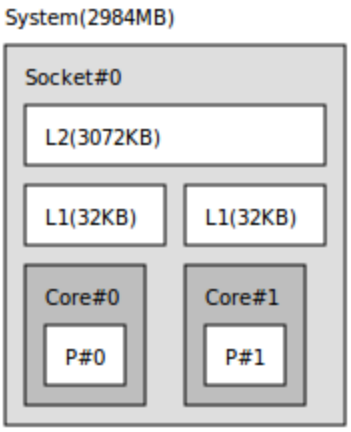
Q1 2012/13

Índex de continguts

I. Entorn d'execució

1. Característiques de l'entorn:

```
Intel(R) Core(TM)2 Duo CPU     P8600  @ 2.40GHz
Microarhitecture:       Core2
Architecture:           x86_64
CPU op-mode(s):         32-bit, 64-bit
CPU(s):                 2
Thread(s) per core:     1
Core(s) per socket:     2
CPU socket(s):          1
NUMA node(s):           1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  23
Stepping:               10
CPU MHz:                2401.000
Virtualization:         VT-x
L1d cache:              32K
L1i cache:              32K
L2 cache:               3072K
clock:                   266MHz
capabilities: fpu fpu_exception wp vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx x86-
64 constant_tsc arch_perfmon pebs bts rep_good aperfmperf pni dtes64 monitor ds_cpl
vmx smx est tm2 ssse3 cx16 xtpr pdcm sse4_1 xsave lahf_lm ida tpr_shadow vnmi
flexpriority cpufreq
```



Sistema Operatiu i compilador:

```
Linux 2.6.32-38-generic #83-Ubuntu SMP x86_64 GNU/Linux
gcc (Ubuntu 4.4.3-4ubuntu5.1) 4.4.3
```

Memòria RAM:

```
Total: 3GiB (bank0: 2GiB, bank1: 1GiB)
description: DIMM DDR Synchronous 800 MHz (1.2 ns)
             width: 64 bits
             clock: 800MHz (1.2ns)
```

Tipus i mida de dades pel compilador GCC:

```
sizeof(char)   =  1 Byte
sizeof(short)  =  2 Bytes
sizeof(int)    =  4 Bytes
sizeof(float)  =  4 Bytes
sizeof(long)   =  8 Bytes
sizeof(double) =  8 Bytes
```

2. Modificació del Makefile

```
--- Makefile.orig 2012-12-26 18:11:26.805492934 +0100
+++ Makefile      2012-12-26 18:13:58.255512876 +0100
@@ -24,7 +24,7 @@
-CC_FLAGS        = -O3 -march=core2 -static
+CC_FLAGS        = $(FLAGS) -O3 -march=core2 -static

@@ -53,6 +53,11 @@
all:        $(PROGRAMS)

+g:
+           make FLAGS='-g'
+
+pg:
+           make FLAGS='-g -pg'
+
+dbg:
+           make CC_FLAGS='$(FLAGS)-march=native -static -pthread -g'
```

3. Condicions de les proves i eines de mesura

Inicialment hem executat el codi i hem realitzat les mesures de temps analitzant les funcions i línies de codi que són més costoses. La compilació ha estat feta amb les següents opcions:

- Per realitzar mesures de Timing (*make*):

  ```
  -O3 -march=native -static -c
  ```

- Per realitzar el profiling amb Gprof (*make pg*):

  ```
  -O3 -march=native -g -pg -static -c
  ```

- Per realitzar el profiling amb Oprofile (*make g*):

  ```
  -O3 -march=native -g -static -c
  ```

- Per “debugar” amb GDB (make dbg)

  ```
  -march=native -g -static -c
  ```

- Per realitzar el timing de l'aplicació s'han introduït crides a clock() entre les seccions comentades amb *"PCA TIMING should start here"* i *"PCA TIMING SHOULD stop here"* escrivint el resultat pel canal d'error.

```
fprintf(stderr,"Elapsed Time: %f sec.\n", (finish - start)/(float)1000000);
```

- El canal stdout ha estat redirigit a /dev/null.
- Al realitzar-se en un portàtil s'ha ajustat sempre la freqüència del processador al màxim possible.
- S'ha utilitzat també la comanda /usr/bin/time que ens ha donat el real time, user time i system time de l'execució.

4. Scripts d'execució

Hem utilitzat un senzill script run.sh que ens ha donat els temps per calcular les estadístiques. S'adjunta als codis del programa.

II. Anàlisis del codi original

1. Timing

El temps de sistema ha resultat ser menyspreable en aquestes proves (Figura 1).

	Elapsed Time (PCA)	Real Time	User Time	Sys Time
2pka.parsed	37,95	40,04	39,98	0,06
1hba.parsed	145,71	146,28	146,21	0,07
4hhb.parsed	133,09	136,88	136,82	0,08
Mitjanes:	*105,58*	*107,74*	*107,67*	*0,07*

Figura 1 - Elapsed Time respecte el fragment objectiu de PCA i sortida de /usr/bin/time. En segons.

2. Profiling

A la Figura 2 mostrem el resultat d'executar ftdock amb l'entrada *2pka.parsed*. Les funcions més costoses són *"electric_field"* i *"pythagoras"* amb un 56,56% i un 14% del temps total consumit respectivament. La funció pythagoras és cridada fins a 1300 milions de vegades.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 57.20     25.50    25.50        1    25.50    31.84  electric_field
 14.24     31.85     6.35 1300937139     0.00     0.00  pythagoras
  3.33     33.34     1.49                              fftw_hc2hc_forward_generic
  2.37     34.39     1.06                              fftw
  2.22     35.38     0.99                              __profile_frequency
  2.13     36.33     0.95                              writev
...
```

Figura 2 - Sortida de GProf pel codi original, anàlisis de funcions, 2pka.parsed

A la Figura 3, resultat de l'execució amb l'entrada *1hba.parsed*, apareixen resultats similars. La funció “electric_field” consumeix un 61% del temps mentre que pythagoras un 15%, aquesta última amb més de 750 milions de crides.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 60.83     98.79    98.79        1    98.79   122.47  electric_field
 14.65    122.59    23.80 750192035     0.00     0.00  pythagoras
  2.72    127.00     4.42                             fftw_executor_simple
  2.69    131.37     4.37                             __profile_frequency
  2.48    135.40     4.03                             writev
  1.92    138.53     3.13                             fftw
  1.19    140.46     1.94    14901     0.00     0.00  gord
...
```

Figura 3 - Sortida de GProf pel codi original, anàlisis de funcions, 1hba.parsed

A l'últim cas, al executar amb l'entrada *4hhb.parsed* obtenim resultats mostrats a la Figura 4.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 67.20    103.89   103.89        1   103.89   128.14  electric_field
 15.75    128.23    24.34 930825049     0.00     0.00  pythagoras
  2.79    131.01     4.28                             __profile_frequency
...
```

Figura 4 - Sortida de GProf pel codi original, anàlisis de funcions, 4hhb.parsed

En quant a les línies de codi més costoses d'aquestes funcions hem observat dos fitxers on es computen les operacions més costoses:

Fitxer electrostatic.c

En aquest fitxer es troba la funció electrostatic_field que es composa de un bucle de complexitat $O(n^5)$. El resultat de gprof linia a linia (gprof --line) ens mostra com diverses línies internes d'aquest bucle són les més costoses. Veiem també com la funció “pythagoras”, segona més costosa, es crida dins d'aquest bucle, Figura 5.

```
for( residue = 1 ; residue <= This_Structure.length ; residue ++ ) {
    for( atom = 1 ; atom <= This_Structure.Residue[residue].size ; atom ++ )
      {
      if( This_Structure.Residue[residue].Atom[atom].charge != 0 ) {
      distance =
            pythagoras( This_Structure.Residue[residue].Atom[atom].coord[1] ,
                        This_Structure.Residue[residue].Atom[atom].coord[2] ,
                        This_Structure.Residue[residue].Atom[atom].coord[3] ,
                        x_centre , y_centre , z_centre ) ;
        if( distance < 2.0 ) distance = 2.0 ;
```

Figura 5 - Fragment del bucle més intern de electrostatic.c

Entre altres trobem els if's, que ens fa pensar en possibles “branch miss-predictions”, la línia que acumula la variable “phi” i que té accessos a memòria, una divisió i una multiplicació, o l'accés a memòria als bucles for com “This_Structure.Residue[residue].size”.

Aquest bucle per tant serà el principal objectiu a optimitzar.

Fitxer coordinates.c

En aquest fitxer trobem la funció “pythagoras”, una de les que hem vist que més es cridava degut al bucle d'”electrostatic_field”. No obstant, aquesta funció realitza només una arrel quadrada, tres multiplicacions i una suma, essent ella sola poc optimitzable.

```
float pythagoras( float x1 , float y1 , float z1 , float x2 , float y2 , float z2 )
{
return sqrt( ( ( x1 - x2 ) * ( x1 - x2 ) )
            + ( ( y1 - y2 ) * ( y1 - y2 ) )
            + ( ( z1 - z2 ) * ( z1 - z2 ) ) ) ;
}
```

Figura 6 - Funció pythagoras a coordinates.c

S'intentarà aplicar alguna tècnica com l'inlining, el buffering, vectorització, etc. si és possible i si és que el compilador no ho fa amb -O3.

3. Speed-Up objectiu

Una vegada vist on és el problema i realitzats els “timing” i “profiling” bàsics podem calcular segons la llei d'Amdahl quin és el màxim “speed-up” que podem obtenir realitzant optimitzacions a les parts del codi més significatives.

Realitzem els càlculs amb el temps ocupat per la rutina electric_field + pythagoras.

SpOpt = Speed-Up de part optimitzada

Popt = % de part optimitzada ~= (57,20+14,24)+(60,83+14,65)+(67,20+15,75)/3 ~= 76,63%

$$SpT = 1 / (1-Popt)+ (Popt/SpOpt)$$

$$SpT = 1 / (1 – 0,76) + (0,76 / SpOpt)$$

Suposem que reduïm a 0 la part que volem optimitzar, llavors aconseguiríem un Speed-Up de:

SpT objectiu: 4,17x.

Evidentment assolir aquest valor no és possible però l'indicador ens serveix per conèixer la millora que com a molt podríem aconseguir centrant-nos en aquestes dues funcions.

III. Optimitzacions

1. Inlining

La primera optimització que portarem a terme serà la de reduir el nombre de crides a pythagoras(). Sabem que el temps de sistema no és molt significatiu i que aquesta millora no suposarà molt rendiment, però introduir el codi en el bucle principal ens permetrà posteriorment fer altres optimitzacions. D'aquesta manera no utilitzarem la directiva “inline” ni crearem cap macro, sinó que directament escriurem el codi de pythagoras, que només és una arrel quadrada, a l'interior del bucle de electrostatic_field.

Fent diverses proves obtenim que:

Speed-up aconseguit = 1,03x

Com era d'esperar hem millorat molt poc el rendiment del codi ja que l'inlining principalment afecta a la part de sistema i en aquest cas el temps de sistema és insignificant.

Un nou anàlisis amb gprof ens mostra que ara electric_field consumeix aproximadament un 74% del temps total, coherent amb Popt calculat a l'apartat anterior.

Les línies de codi de electrostatics.c que més consumeixen no han variat respecte l'anàlisis anterior.

2. Buffering de SQRT

Hem comprovat que no és possible realitzar aquesta optimització ja que els valors que es passen a pythagoras són floats que representen coordenades i són sempre diferents per cada input.

3. BitHacks a SQRT - Fast Inverse Square Root (Quake 3)[1]

Aquest mètode es basa en realitzar una nova implementació de la funció de l'arrel quadrada de la biblioteca math.c. Per fer-ho ens basem en la implementació de J.Carmack de Id Software que va realitzar al Quake 3. Es basa en aproximacions de Newton a successives arrels.

Creem per aixo una nova funció anomenada f_inv_sqrt() a electrostatic.c, amb el contingut següent:

```
float f_inv_sqrt(float x)
{
  float xhalf = 0.5f * x;
  int i = *(int*)&x;           //store floating-point bits in integer
  i = 0x5f375a86 - (i >> 1);  // initial guess for Newton's method
  x = *(float*)&i;             // convert new bits into float
  x = x*(1.5f - xhalf*x*x);    // One round of Newton's method
  return 1/x;
}
```

Canviant la crida de la original pythagoras(), obtenim un resultat negatiu. El temps d'execució ha passat a ser:

	Elapsed Time (PCA)	*Real Time*	*User Time*	*Sys Time*
2pka.parsed	37,95	40,04	39,98	0,06
2pka.parsed (f_inv_sqrt)	43,49	45,58	45,54	0,04

Per tant hem empitjorat el temps en un 14%. L'explicació d'això es senzilla, i es que la implementació de la biblioteca math de GNU C es molt eficient i fins i tot es fan servir operacions de processador com sqrtss o sqrtsd i vectorització.

+codi: https://github.com/andikleen/glibc/blob/master/math/s_csqrt.c

1+*info: http://betterexplained.com/articles/understanding-quakes-fast-inverse-square-root/*

4. Utilització de float vs double

A la versió original la crida a sqrt és a double sqrt(double x). A n'aquesta funció però se li passen dos valors de tipus float que tenen 4 bytes enlloc de 8 en aquesta màquina i per tant estem desaprofitant memòria. Al mirar el codi compilat amb objdump -D -M intel, veiem que si no apliquem -O3 i degut a -march=core2, el codi és vectoritzat usant la instrucció de doble precisió:

```
sqrtsd xmm0,xmm1
```

Per altra part si modifiquem la crida i ho fem a float sqrtf(float x), no perdrem precisió i el resultat de la compilació serà que s'utilitzarà la instrucció de precisió simple obtenint un millor rendiment:

```
sqrtss xmm0,xmm1
```

No obstant, al compilar amb -O3 s'elimina aquesta millora ja que el propi compilador aplica la optimització.

5. Eliminant salts condicionals

En la comparació de distance amb “2.0” es realitza un if innecessari que eliminem. Això ens millora el codi compilat eliminant algunes operacions.

```
if( distance < 2.0 ) distance = 2.0 ;
                        if( distance >= 2.0 )            <- if sobrant
                          {
                            if( distance >= 8.0 )
                            ...
                            phi +=( This_Structure.Residue[residue].Atom[atom]
                                          .charge / ( epsilon * distance ) ) ;
                          }
```

La millora obtinguda és de 37,18s sobre 40,04s, per tant:

Speed-up = 1,08x

6. Eliminació de càlculs repetits i accessos a memòria distants

En el bucle principal de complexitat O(n^5) veiem que es realitzen, per tota combinació de coordenades centrals y_centre, x_centre i y_centre, els càlculs de si un àtom té càrrega neutra o no. A més l'accés a memòria de les dades per aquests àtoms no és consecutiu ja que s'accedeix cada vegada a posicions molt distants i no consecutives de la memòria per la naturalesa de les estructures “Structure”, “Amino_Acid” i “Atom”.

Com a solució creem una nova estructura anomenada “ch_atom” que contindrà la càrrega d'un àtom i les seves tres posicions X, Y i Z. Es crearà un vector d'aquesta estructura i s'emmagatzemaran abans d'entrar al bucle principal els valors de cada àtom amb càrrega no nula.

D'aquesta manera aconseguim dues coses:

1. Evitar accedir a posicions molt distants de memòria al bucle de complexitat O(n^5)
2. Reduïr la complexitat del bucle a O(n^4)

Al analitzar el resultat podem observar que la millora en el rendiment és notable. Fent la mitja dels temps obtinguts en tots els jocs de proves i comparant-los amb els de Figura 1 obtenim aquesta vegada un **Speed-up = 1,93x** respecte l'original.

7. Vectorització de SQRT

Veiem que la funció sqrtf() és vectoritzable degut a que podriem realitzar les operacions de “[x,y,z] – [x,y,z]_centre” i els quadrats d'aquests resultast de diversos àtoms al mateix temps.

En l'apartat anterior hem escrit la línea de codi aprofitant l'estructura realitzada:

```
distance =  sqrtf((
   (charged_atoms[atom].x - x_centre) * (charged_atoms[atom].x - x_centre))
+ ((charged_atoms[atom].y - y_centre) * (charged_atoms[atom].y - y_centre))
+ ((charged_atoms[atom].z - z_centre) * (charged_atoms[atom].z - z_centre))
) ;
```

No obstant, aquesta estructura no ens serveix ja que les dades estan en mode “interleaved” (xyzxyzxyz...) i les necessitem en mode “plannar” (xxxyyyzzz...).

Per això desfarem l'estructura de l'apartat anterior i ho convertirem en arrays de floats independents, alineant-los a més a 16 bytes i per tant havent d'afegir padding en el cas de les coordenades. Il·lustrem la operació de vectorització que realitzarem:

coords$_i$: [x_1 y_1 z_1 # x_2 y_2 z_2 # x_3 y_3 z_3 # x_4 y_4 z_4 #...] (# padding)

center: [x_center, y_center, z_center, #]

charges$_i$: [c_1 c_2 c_3 c_4 c_5 c_6 ...]

D'aquesta manera posem dins un "__m128 coord" un total de 3 floats i un padding, i dins "centers" els centers i un padding. Restem vectorialment "coord – centers". Llavors multipliquem vectorialment el resultat per ell mateix i ja tenim dins un __m128 els quadrats de les distàncies de x_i ,y_i i z_i . Llavors només cal extreure aquests valors i aplicar-hi la SQRTF.

L'Speed-up que hem obtingut respecte l'original amb aquesta mesura és de:

Speed-up = 1,93x

Hem detectat que el compilador ja genera una vectorització dels bucles i d'aquest codi i hem vist com l'Speed-up aconseguit manualment en aquest cas és igual al que aconsegueix el propi compilador. Creiem per tant que no cal insistir més en aquesta tècnica.

8. Threads

Realitzarem la versió amb fils d'execució del programa sobre el bucle que realitza tot el treball del grid_size, és a dir, el primer for.

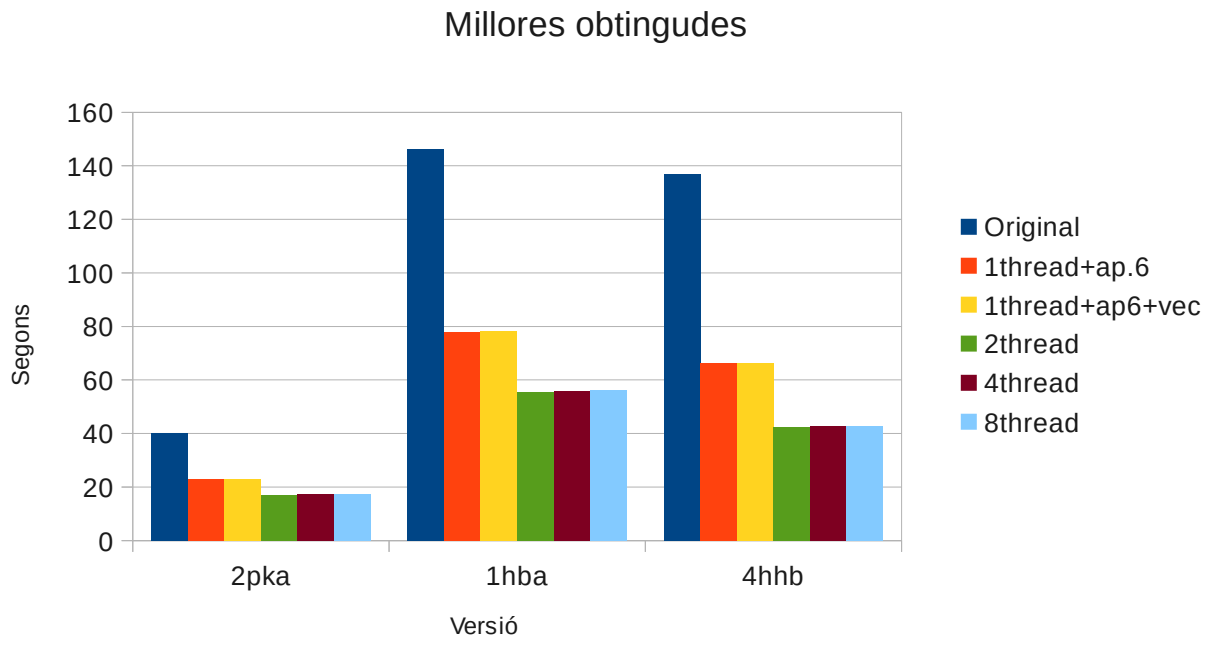
En mitjana hem obtingut unes mesures de temps de 38,11s vs els 107,74 segons de l'original. Això ens aporta un speed-up de:

Speed-up = 2,82x

En aquesta versió hem dividit equitativament l'espai de treball en parts iguals per cada thread. Gràcies a la caché de L2 de 3MB i les individuals de 32KB hem pogut observar una millora en el rendiment amb aquesta organització. Tot i així amb un profiling hem descobert que el que relantitzava ara el codi era l'accés al vector coords a cada un dels threads. El vector és global i accedit de forma seqüencial i per tant hi cap una bona part a la caché de

L2, tot i que s'ha de copiar a les L1 individuals. Aquestes copies i fan que es ralentitzi l'execució.

Per altra banda el nombre de threads òptim ha estat de 2, ja que a la màquina no disposem d'hyper-threading i només hi ha dos nuclis.



IV. Recursos

El desenvolupament del codi s'ha realitzat a un repositori de codi de Google amb llicència GPLv3, disponible a:

https://code.google.com/p/pca-fib/