# Projecte Final: Protein Docking
# PCA

Jordi Romero de Villalonga

16 de Juny de 2010

# Índex

# Capítol 1

# Introducció

Aquest projecte consta d'optimitzar una aplicació que té com a objectiu principal buscar **acoblaments rígids entre parells de biomolècules**. Aquest algorisme es basa en aplicar transformacions de Fourier mitjançant la llibrería **FFTW**.

L'objectiu del projecte és reduir el temps d'execució real (a partir d'ara *elapsed time*) entre dos punts concrets del codi. Aquests són al voltant del codi que efectua les següents tasques:

- Llegir les estructures mol·leculars dels fitxers d'entrada
- Carregar a la memòria aquestes estructures
- Calcular la graella que representarà l'espai
- Crear els plans per a les transformacions de Fourier
- Calcular el camp elèctric en cada punt de la graella

Posteriorment a això, el programa originalment prova d'encaixar en totes les posicions possibles les dues mol·lècules, però per reduir el temps d'execució es va simplificar.

Concretament, com veurem a continuació, el gruix de l'elapsed time es perd en el càlcul del camp elèctric per a tota la graella tridimensional. Aquest el que fa és recórrer una matriu de tres dimensions i a cada punt calcular la suma de camps electrics respecte cadascun dels àtoms de cada una de les mol·lècules d'entrada. Com es pot observar, es tracta d'una operació realitzada dins d'un bucle de 5 nivells.

# Capítol 2

# Anàlisi del programa

Després de compilar la llibrería **FFTW** i el programa principal (**3D_Dock**) ens disposem a analitzar el comportament d'aquest a partir de *profiling*.

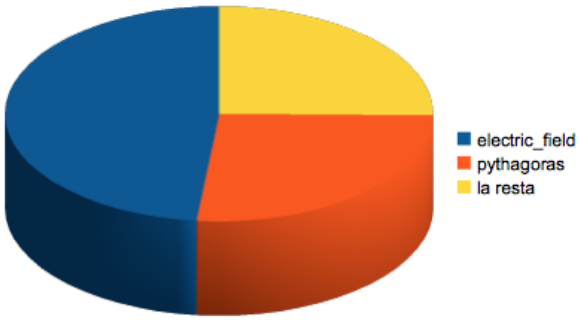
## 2.1 Profiling

La primera conclusió que treiem és immediata: hi ha dos grans funcions que ocupen juntes la major part de l'*elapsed time*:

- `electric_field` 48.3 %
- `pythagoras` 26.5 %

A continuació es pot observar com aquesta distribució marca condundentment el nostre principal objectiu a optimitzar.

**Figura 2.1: Distribució del l'elapsed time en crides a funcions**



Observant el codi, s'aprecia que la funció `pythagoras` es crida des d'`electric_field`, per tant tenim que el 75% del temps d'execució s'ocupa calculant el camp electric en cada punt d'aquesta graella.

## 2.2 Anàlisi

El codi de la funció `electric_field` és prou senzill com per mostrar-lo (simplificat) a continuació:

**Codi 2.1: Codi original d'electric_field**

```
1  /* Inicialitzacions */
2
3  for( x = 0 ; x < grid_size ; x ++ )
4  {
5    for( y = 0 ; y < grid_size ; y ++ )
6    {
7      for( z = 0 ; z < grid_size ; z ++ )
8      {
9        phi = 0 ;
10       for(residue = 1 ; residue <= This_Structure.length; residue++)
11       {
12         for(atom=1; atom <= This_Structure.Residue[residue].size;
               atom++)
13         {
14           if(This_Structure.Residue[residue].Atom[atom].charge != 0)
15           {
16             distance = pythagoras(
17               This_Structure.Residue[residue].Atom[atom].coord[1],
18               This_Structure.Residue[residue].Atom[atom].coord[2],
19               This_Structure.Residue[residue].Atom[atom].coord[3],
20               x_centre , y_centre , z_centre
21             ) ;
22
23             if( distance < 2.0 ) distance = 2.0 ;
24             if( distance >= 2.0 )
25             {
26               if( distance >= 8.0 )
27                 epsilon = 80 ;
28               else
29                 if( distance <= 6.0 )
30                   epsilon = 4 ;
31                 else
32                   epsilon = ( 38 * distance ) - 224 ;
33               phi += (This_Structure.Residue[residue].Atom[atom].
                     charge / (epsilon * distance));
34             }
35           }
36         }
37       }
38       grid[gaddress(x,y,z,grid_size)] = (fftw_real)phi;
39     }
40   }
41 }
```

El significat d'aquest codi és el següent:

- Per cada punt de la graella tridimensional (línia 7):

- Recorrem totes les mol·lècules que hem llegit i per cadascuna d'elles (10):
- Recorrem els àtoms d'aquesta, i a cada àtom (12):
- En comprovem la càrrega. Si no és nul·la (14):
- Calculem la distància entre l'àtom i el punt de la graella on estavem (16)
- Calculem el camp elèctric provocat per l'àtom en aquest punt (37)
- Finalment acumulem aquest càlcul final a la matriu de camps elèctrics (42)

## 2.3 Temps d'execució

Com hem comentat prèviament, es mesura només el temps d'execució real entre dos punts del programa. Aquests venien pre-fixats en l'enunciat. Per mesurar el temps d'execució prenem dos captures de l'instant del rellotge del sistema, per tant mesurem la diferència de temps entre els dos instants amb precisió de $\mu$ segons:

**Codi 2.2: Mesura del temps d'execució real**

```
1 struct timeval start, end;
2 gettimeofday(&start, NULL);
3 /* ... timed code ... */
4 gettimeofday(&end, NULL);
5 float elapsed = (end.tv_sec + (end.tv_usec / 1000000.0)) -
6                 (start.tv_sec + (start.tv_usec / 1000000.0));
7 fprintf(stderr, "%f\n", elapsed);
```

Amb això tenim que el programa treurà pel canal 2 (sortida d'error estàndard) l'*elapsed time*. El nostre script d'executar el programa llegirà aquest valor per calcular-ne el temps promig.

En una primera execució del programa, sense haver fet encara cap modificació excepte l'instrumentació per capturar el temps d'execució real entre els dos punts esmentats, obtenim els següents resultats:

**Figura 2.2: Temps d'execució del programa original - segons**

| | test1 | test2 | test3 | total |
|---|---|---|---|---|
| **Versió original** | 45.5 | 174.5 | 164.3 | 384.3 |

Cada execució que es mostri a partir d'ara, a l'igual que aquesta, serà el resultat d'executar per a una versió específica del programa amb els tres jocs de proves proporcionats i iterant 5 vegades cada prova. Els valors mostrats són les mitjanes d'aquestes execucions. Quan parlem de guany, sempre ho farem respecte a aquests temps d'execució. Ens referirem al guany global com a diferència de temps **total**, per tant transversal als tres jocs de proves. Quan parlem del millor dels casos serà amb un joc de proves individual, que pugui donar uns resultats diferents al valor global.

# Capítol 3

# Optimitzacions

Detectem que els accessos a dades que resideixen a la memòria dins la funció `electric_field` no són òptims, com comentem a continuació. Aquest és doncs un motiu que genera més tràfic de dades entre línies de cache de l'imprescindible. També observem que dins de `pythagoras` s'efectuen operacions aritmètiques costoses com són productes i arrel quadrada en operands de coma flotant. Aquests seran els principals punts d'atac de les nostres optimitzacions, així com intentar paral·lelitzar aprofitant les capacitats dels processadors actuals.

## 3.1 Accessos a memòria

Observem com s'emmagatzemen les estructures de dades a les que s'accedeix, i veiem que hi ha un array de mol·lècules que conté un array d'àtoms (entre d'altres camps) i aquest conté les seves coordenades i càrrega. Aquestes dades són les úniques que consultem (coordenades i càrrega) però estan rodejades de més informació inútil.

Concretament, consultant les estructures de dades de `structures.h` observem que per cada mol·lècula (amb $n$ àtoms) estem utilitzant $n * 36 + 24$ bytes, quan només en necessitem $n * 16$, o sigui, 4 valors de coma flotant (3 coordenades i la càrrega).

També observem en el codi de la figura 2.1 que per cada àtom, només realitzem els càlculs i modifiquem el resultat si aquest té càrrega elèctrica. Això significa que per cada punt de la graella tridimensional estem accedint i comprovant els valors d'uns àtoms que MAI utilitzem.

Per solucionar aquests problemes decidim copiar les dades que necessitem en posicions consecutives per optimitzar la jerarquia de memòria. Copiarem les coordenades i la càrrega de cada àtom, però només d'aquells amb càrrega elèctrica. Això ho fem al principi de la funció `electric_field` d'aquesta manera:

**Codi 3.1: Còpia de dades per optimitzar jerarquia de memòria**

```
1 /* cnt = sum of atoms in all the molecules */
2 unsigned int cnt = 0;
```

```
3  for( residue = 1 ; residue <= This_Structure.length ; residue ++ )
4    cnt += residue * This_Structure.Residue[residue].size;
5
6  unsigned int atoms = 0;
7  float *atom_coords, *aux_coord;
8  float *atom_charges, *aux_charge;
9
10 atom_coords = malloc(cnt * 3 * sizeof(float));
11 atom_charges = malloc(cnt * sizeof(float));
12
13 for( aux_coord = atom_coords, aux_charge = atom_charges, residue = 1
       ; residue <= This_Structure.length ; residue ++ )
14 {
15   for( atom = 1 ; atom <= This_Structure.Residue[residue].size ;
         atom ++ )
16   {
17     if( This_Structure.Residue[residue].Atom[atom].charge != 0 )
18     {
19       *aux_coord = This_Structure.Residue[residue].Atom[atom].coord
             [1];
20       *(aux_coord + 1) = This_Structure.Residue[residue].Atom[atom].
             coord[2];
21       *(aux_coord + 2) = This_Structure.Residue[residue].Atom[atom].
             coord[3];
22       aux_coord += 3;
23       *aux_charge = This_Structure.Residue[residue].Atom[atom].
             charge;
24       aux_charge++;
25       atoms++;
26     }
27   }
28 }
29
30 /* atoms = number of atoms with charge */
```

Com es pot veure, creem dos *arrays* de *floats* i només hi afegim els que tenen una càrrega positiva. Això a part de reduir l'espai de memòria que mourem entre línies de cache, permet fer menys iteracions en els bucles més interns. De fet, un cop feta aquesta reestructuració de les dades a la memòria, podem canviar radicalment els dos bucles interns de la funció per ignorar la diferenciació entre mol·lècules i àtoms, ja que no ens afecta, i simplement iterar pels àtoms de totes les mol·lècules acumulant els valors del camp elèctric en el punt en qüestió: ara només fem un bucle de `atoms` iteracions, o sigui el número d'àtoms de totes les mol·lècules amb càrrega elèctrica, i que tenim emmagatzemats en posicions consecutives en els *arrays* `atom_coords` i `atom_charges`.

**Codi 3.2: Bucle intern que substitueix els anteriors 2 bucles més interns**

```
1 /* ... */
2 aux_coord = atom_coords;
3 aux_charge = atom_charges;
4 for ( atom = 0; atom < atoms; atom++)
5 {
6   distance = pythagoras( *aux_coord , *(aux_coord+1) , *(aux_coord
        +2) , x_centre , y_centre , z_centre ) ;
7   /* ... */
```

Tot i que en una versió posterior del programa s'ha canviat lleugerament la manera d'emmagatzemar aquestes dades, ho deixem per més endavant. La

versió de la que s'han près mesures de temps i s'ha analitzat és la que s'ha vist ara.

Tot i que aquests canvis es fan per millorar l'aprofitament de la jerarquia de memòria i reduir iteracions en els bucles interns, l'objectiu final és també situar les dades d'una manera que es puguin explotar altres tècniques com veurem més endavant.

Els resultats d'aplicar només aquests canvis ja mostren que val la pena replantejar l'estratègia d'emmagatzemament, com es veu en el resultat següent:

**Figura 3.1: Resultats amb una millor organització de les dades en memòria**

| | test1 | test2 | test3 | total |
|---|---|---|---|---|
| Versió original | 45.5 | 174.5 | 164.3 | 384.3 |
| **Optimització 1** | 31.2 | 119.0 | 104.9 | 255.1 |

Com es pot deduir dels números de la figura 3.1, obtenim un guany global de **1.51**, i en el millor dels casos de **1.57**.

## 3.2 Vectorització

### 3.2.1 Vectorització pythagoras

Després de reorganitzar les dades a la memòria, observem mitjançant *profiling* que més d'una quarta part del temps perdut a `electric_field` es perd realitzant el càlcul de la distància entre dos punts mitjançant la crida a `pythagoras`. Per solucionar això, decidim convertir aquest càlcul en operacions dins de la mateixa funció `electric_field` i paral·lelitzar-ne les operacions aritmètiques mitjançant vectorització.

Per tal de poder aplicar vectorització en només una part del bucle intern, decidim separar aquest en dos bucles amb el mateix número d'iteracions, on el primer calcularà les distàncies per tots els àtoms i el segon, a partir d'aquest càlcul, calcularà el camp elèctric. Aquesta estratègia la seguirem utilitzant més endavant. Com que el cos del bucle té un cost computacional altíssim, l'*overhead* afegit per separar-lo en dos bucles és totalment menyspreable, i en canvi les optimitzacions que això ens permet fan que aquest petit sacrifici surti molt recompensat.

El que fem doncs és substituir la crida a `pythagoras` per les operacions que aquesta funció efectua (suma els quadrats de les diferències entre components dels dos punts i després d'això en fa l'arrel quadrada) per operacions vectorials que permeten paral·lelitzar-les.

El primer problema amb el que ens trobem és que per tal de poder fer servir operacions vectorials amb les coordenades que nosaltres hem guardat consecutivament a memòria, necessitem afegir un padding d'un byte per cada conjunt de tres coordenades, per tal de tenir alineats a 16 bytes els conjunts de coordenades. Això ens fa utilitzar una mica més d'espai però és imprescindible per fer accessos

alineats a cada àtom. La modificació per això és trivial. També cal pensar en reservar la memòria alineada a 16 bytes mitjançant la funció `posix_memalign`. Per altra banda, ara el càlcul de la distància també el farem per a tots els àtoms, de manera que primer establim les distàncies per tots els àtoms i a continuació en calcularem el camp elèctric:

**Codi 3.3: Primer dels dos bucles interns aplicant vectorització**

```
1  aux_distance = atom_distances;
2  __m128 _centers = _mm_setr_ps(x_centre, y_centre, z_centre, 0.0);
3  for ( atom = 0; atom < atoms; atom++)
4  {
5    __m128 *coords = (__m128*) aux_coord;
6    __m128 aux = _mm_sub_ps(*coords, _centers);
7    aux = _mm_mul_ps(aux, aux);
8    *aux_distance = sqrt( *((float*) &aux) + *((float*) (&aux)+1) +
         *((float*) (&aux)+2) );
9    aux_coord += 4;
10   aux_distance++;
11 }
```

En aquest cas utilitzem operacions vectorials (que operen de 4 en 4 *floats*) tot i que només fem cas dels tres primers. En aquest fragment del codi doncs només obtenim una paral·lelització de 3 en 3, però la millora ja s'aprecia lleugerament:

**Figura 3.2: Resultats amb una millor organització de les dades en memòria**

| | test1 | test2 | test3 | total |
|---|---|---|---|---|
| Versió original | 45.5 | 174.5 | 164.3 | 384.3 |
| Optimització 1 | 31.2 | 119.0 | 104.9 | 255.1 |
| **Optimització 2** | 27.8 | 107.4 | 91.6 | 226.8 |

En aquest cas l'**speedup** global respecte el codi original és de **1.69** tot i que en el millor dels casos és de **1.79**.

### 3.2.2 Unroll i vectorització de sqrt

Un cop vectoritzades les operacions auxiliars del cos de `pythagoras`, i aprofitant el separament de bucles fet anteriorment, apliquem un unrolling de grau 4 per tal de poder calcular paral·lelament mitjançant operacions vectorials l'operació d'arrel quadrada que és molt costosa.

El bucle que hem mostrat a l'apartat anterior acaba amb aquest aspecte:

**Codi 3.4: Unrolling i vectorització de pythagoras**

```
1  aux_distance = atom_distances;
2  __m128 _centers = _mm_setr_ps(x_centre, y_centre, z_centre, 0.0);
3  for ( atom = 0; atom < atoms-3; atom += 4)
4  {
5    __m128 pyths;
6    __m128 aux = _mm_sub_ps(*((__m128*) aux_coord), _centers);
7    aux = _mm_mul_ps(aux, aux);
```

```
8   *((float*) &pyths) = *((float*) &aux) + *((float*) (&aux)+1) + *((
        float*) (&aux)+2);
9   aux_coord += 4;
10
11  aux = _mm_sub_ps(*((__m128*) aux_coord), _centers);
12  aux = _mm_mul_ps(aux, aux);
13  *((float*) &pyths+1) = *((float*) &aux) + *((float*) (&aux)+1) +
        *((float*) (&aux)+2);
14  aux_coord += 4;
15
16  aux = _mm_sub_ps(*((__m128*) aux_coord), _centers);
17  aux = _mm_mul_ps(aux, aux);
18  *((float*) &pyths+2) = *((float*) &aux) + *((float*) (&aux)+1) +
        *((float*) (&aux)+2);
19  aux_coord += 4;
20
21  aux = _mm_sub_ps(*((__m128*) aux_coord), _centers);
22  aux = _mm_mul_ps(aux, aux);
23  *((float*) &pyths+3) = *((float*) &aux) + *((float*) (&aux)+1) +
        *((float*) (&aux)+2);
24  aux_coord += 4;
25
26  *((__m128*) aux_distance) = _mm_sqrt_ps(pyths);
27  aux_distance +=4 ;
28 }
29
30 for ( ; atom < atoms; atom++)
31 {
32   __m128 aux = _mm_sub_ps(*((__m128*) aux_coord), _centers);
33   aux = _mm_mul_ps(aux, aux);
34   *aux_distance = sqrt( *((float*) &aux) + *((float*) (&aux)+1) +
        *((float*) (&aux)+2) );
35   aux_coord += 4; aux_distance++;
36 }
```

Com es pot observar, l'unroll crea la necessitat d'un segon bucle per assegurar que quan `atoms` no és multiple de 4 també es processin tots els àtoms. El que s'ha canviat és que ara es processen quatre àtoms, i finalment la operació d'arrel quadrada s'aplica a les distàncies dels quatre àtoms alhora mitjançant l'operació vectorial `_mm_sqrt_ps`. La resta segueix igual. Aquest canvi ens deixa amb els següents resultats:

**Figura 3.3: Resultats amb una millor organització de les dades en memòria**

| | test1 | test2 | test3 | total |
|---|---|---|---|---|
| Versió original | 45.5 | 174.5 | 164.3 | 384.3 |
| Optimització 1 | 31.2 | 119.0 | 104.9 | 255.1 |
| Optimització 2 | 27.8 | 107.4 | 91.6 | 226.8 |
| **Optimització 3** | 22.2 | 84.2 | 68.3 | 174.7 |

Amb aquestes millores el nostre programa ja funcioni més del doble de ràpid que l'original, amb un guany global de **2.2** i en el cas millor de **2.41**.

### 3.2.3 Unroll i vectorització del càlcul de phi

Anàlogament, apliquem la mateixa tècnica en un altre càlcul costós que estem realitzant per cada àtom: el camp elèctric provocat.

Com que tenim el càlcul que implica multiplicacions i divisions, decidim paral·lelitzarles mitjançant vectorització, així que també apliquem unrolling de grau 4:

**Codi 3.5: Unrolling i vectorització de phi**

```
1  aux_distance = atom_distances;
2  aux_charge = atom_charges;
3  for ( atom = 0; atom < atoms; atom += 4)
4  {
5    /* ... if's pels quatre atoms: definim epsilon1, 2, 3 i 4 ... '*/
6    __m128 epsilons = _mm_setr_ps( epsilon1, epsilon2, epsilon3,
         epsilon4);
7    epsilons = _mm_mul_ps(epsilons, *((__m128*) aux_distance));
8    epsilons = _mm_div_ps(*((__m128*) aux_charge), epsilons);
9    phi += (*((float*) &epsilons) + *((float*) (&epsilons)+1) + *((
         float*) (&epsilons)+2) + *((float*) (&epsilons)+3));
10   aux_charge += 4; aux_distance += 4;
11 }
12 for (; atom < atoms; atom++)
13 {
14   if( *aux_distance < 2.0 ) *aux_distance = 2.0 ;
15   if( *aux_distance >= 8.0 )
16     epsilon = 80 ;
17   else
18     if( *aux_distance <= 6.0 )
19       epsilon = 4 ;
20     else
21       epsilon = ( 38 * *aux_distance ) - 224 ;
22   phi += ( *aux_charge / ( epsilon * (*aux_distance) ) ) ;
23   aux_charge++; aux_distance++;
24 }
```

En el codi de la figura 3.5 es realitzen exactament les mateixes operacions en els dos bucles, amb la diferència que en el primer (a part de que s'han eliminat els if's per compactar el codi en aquest document) s'opera sempre de quatre en quatre. Aquest paral·lelisme que ens ofereix la vectorització permet assolir uns temps d'execució inferiors, com podem observar:

**Figura 3.4: Resultats amb una millor organització de les dades en memòria**

| | test1 | test2 | test3 | total |
|---|---|---|---|---|
| Versió original | 45.5 | 174.5 | 164.3 | 384.3 |
| Optimització 1 | 31.2 | 119.0 | 104.9 | 255.1 |
| Optimització 2 | 27.8 | 107.4 | 91.6 | 226.8 |
| Optimització 3 | 22.2 | 84.2 | 68.3 | 174.7 |
| **Optimització 4** | 16.1 | 60.6 | 43.9 | 120.6 |

Aquests temps, traduits a **speedup**, tenim que en el cas global estem executant el codi **3.19** vegades més ràpid, i en el cas millor **3.74**.

A part dels canvis d'unroll bàsic i el posterior ús d'instruccions vectorials, s'han reordenat les operacions que podíen causar dependències, però com que la millora no ha estat molt notable no es contemplen individualment. El resultat però inclou aquestes reordenacions.

## 3.3 Paral·lelització amb threads

Per tal d'aprofitar al màxim les característiques dels processadors actuals, cal fer que el programa sigui capaç de dividir-se càrrega de treball entre els diferents nuclis de procés de que disposa l'ordinador. Per fer això usarem threads mitjançant la implementació **pthreads**.

De cara a poder usar aquesta tècnica, hem extret de la funció `electric_field` tota la part corresponent als bucles en una nova funció: `electric_field_partial`. D'aquesta manera podem des de la funció principal inicialitzar les dades que necessitem com ara la còpia de les coordenades dels àtoms o les seves càrregues en vectors compactes i alineats a 16 bytes, etc. Les variables que necessitem que la segona funció llegeixi les fem globals. En aquest cas la funció que volem paral·lelitzar **només modificarà una variable global**, i per cada punt de la graella **accedirà a una posició diferent**, per tant **no cal usar mutex** per garantir que no hi ha conflictes en aquests accessos.

El codi resultant de la funció principal on cridem a la funció auxiliar que farà tot el càlcul és el seguent:

**Codi 3.6: Creació de threads**

```
1  pthread_t threads[NT];
2  int rc, t;
3
4  for(t = 0; t < NT; t++)
5  {
6    rc = pthread_create(&threads[t], NULL, electric_field_partial, (
         void *) t);
7    if (rc)
8    {
9      printf("ERROR CREATING THREAD (%d) = %d\n", t, rc);
10     exit(-1);
11   }
12 }
13
14 for(t = 0; t < NT; t++)
15 {
16   if (rc = pthread_join(threads[t], NULL))
17   {
18     printf("ERROR JOINING THREAD (%d) = %d\n", t, rc);
19     exit(0);
20   }
21 }
```

Com es pot veure, passem un únic paràmetre que és l'identificador del thread, perquè així a la funció auxiliar es pugui saber quin fragment de les dades ha de processar. Hem decidit dividir la càrrega de treball a partir de separar els fragments del bucle més extern a processar. D'aquesta manera tots els threads tindran una càrrega de treball igual, ja que independentment del punt on es

processi cal fer els mateixos càlculs. El codi que discrimina això a la funció auxiliar és el següent:

**Codi 3.7: Divisió de feina a cada thread**

```
1 int x_from, x_to;
2 x_from = tid * (g_grid_size / NT);
3 x_to = (tid+1) * (g_grid_size / NT);
4 if (tid == NT - 1) x_to = g_grid_size;
5
6 for( x = x_from ; x < x_to ; x ++ )
7 {
8   printf( "." ) ;
9   /* ... tot segueix igual ...*/
```

On NT és el número de threads, que s'ha fixat en 16 ja que és el valor que millor resultats ha mostrat a diferents màquines, tot i que la millora respecte un número més baix com 4 és molt poc notable en un processador Core 2 Duo, en un Core i7 amb quatre cores i hyper-threading la millora sí que es nota molt.

El resultat final del programa és el que podem mesurar ara, i mostrem la taula completa de temps d'execució a continuació:

**Figura 3.5: Resultats amb una millor organització de les dades en memòria**

| | test1 | test2 | test3 | total |
|---|---|---|---|---|
| Versió original | 45.5 | 174.5 | 164.3 | 384.3 |
| Optimització 1 | 31.2 | 119.0 | 104.9 | 255.1 |
| Optimització 2 | 27.8 | 107.4 | 91.6 | 226.8 |
| Optimització 3 | 22.2 | 84.2 | 68.3 | 174.7 |
| Optimització 4 | 16.1 | 60.6 | 43.9 | 120.6 |
| **Optimització 5** | 11.6 | 45.2 | 28.0 | 84.8 |

En la darrera optimització, el guany global és de **4.53**, tot i que si ens quedem només amb el test 3 el guany observat és de **5.87**.

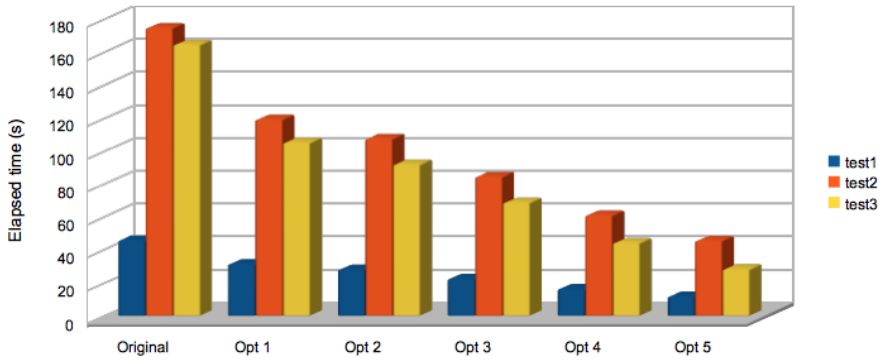
# Capítol 4

# Conclusions

Com hem anat veient en els diferents apartats dins el capítol d'Optimitzacions, el temps d'execució real ha anat disminuit més o menys linealment. Les optimitzacions s'han anat introduint esglaonadament, de manera que no hi ha hagut cap canvi sobtat, tot i que les dues darreres han estat les que han tingut més impacte ja que s'ha pogut paral·lelitzar moltes operacions.

A la figura 4.1 podem veure la progressió de l'elapsed time per cadascun dels tres tests:

**Figura 4.1: Elapsed time per cadascun dels tres tests en tots els passos**




Aquesta retallada de temps d'execució es converteix en un **speedup** global (o sigui, executant els tres jocs de proves) de **4.53**, tot i que si ens quedem només amb el joc de proves tres, l'speedup entre la versió original i la final és de **5.87**.

A la figura 4.2 mostrem l'evolució de l'speedup amb els diferents jocs de proves:

Tot i que com hem comentat, si ens quedem només amb el joc de proves número tres, encara s'aprecien millor resultats, com es pot apreciar a la figura 4.3:

**Figura 4.2: Speedup global per a totes les versions del programa**

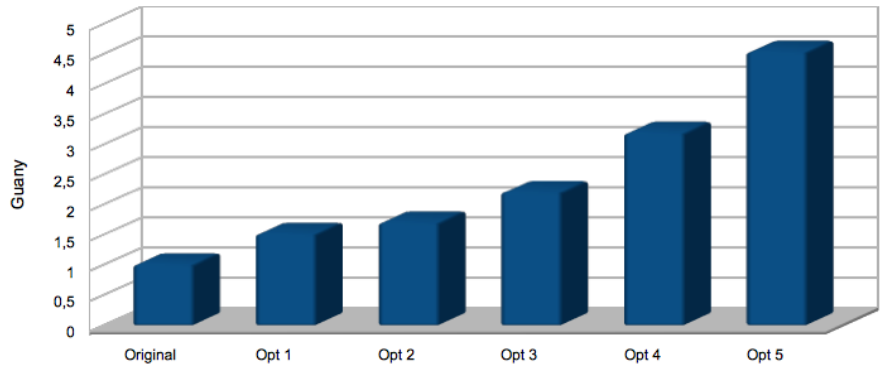



**Figura 4.3: Speedup de test3 per a totes les versions del programa**