Projecte final: Protein Docking

Genís Riera Pérez
Iulian Valentín Brumar

Programació Conscient de l'Arquitectura
Q2 - 2012/2013

Índex

1. Consideracions prèvies

- Totes les versions del programa han estat compilades amb la versió **(SUSE Linux) 4.6.2** del compilador **gcc**.

- La tasca de compilar les diverses versions del programa amb diferents opcions del compilador en funció dels nostres interessos (mesura dels temps, fer *profiling*, *debugging*, etc.) s'ha automatitzat amb un fitxer `Makefile` adjunt al comprimit de la pràctica. Per a conèixer el seu ús consulteu el fitxer `README.txt` adjunt al comprimit de la pràctica.

- Totes les execucions han estat realitzades sobre l'entorn descrit a l'annex A.

- Al llarg del document es mencionen 3 jocs de proves que s'han fet servir per provar totes les versions del programa, que anomenem test 1, test 2 i test 3. Aquests referencien els següents fitxers d'entrada:

  - `test 1: -static 2pka.parsed -mobile 5pti.parsed`
  - `test 2: -static 1hba.parsed -mobile 5pti.parsed`
  - `test 3: -static 4hhb.parsed -mobile 5pti.parsed`

- Per a generar tots els anàlisis amb les diferents eines de *profiling* utilitzades s'ha executat el programa amb el test 1 i redireccionant la sortida a `/dev/null`.

- Per garantir que cada versió del programa no altera la correctesa de la versió original, a cada optimització implementada hem comparat la seves sortides (una per cadascun dels 3 tests), amb les originals. Aquest procés l'hem realitzat sempre, abans de començar a fer mesures de temps i analitzar resultats. Totes les optimitzacions mostrades en aquest document satisfan la correctesa del programa. Addicionalemnt, dins del directori `sortides_programa_opt_&_orig` adjunt al comprimit de la pràctica es poden trobar les sortides generades per la versió més òptima i l'original del programa (una sortida per cadascun dels 3 tests), per a que es pugui demostrar que la versió més òptima funciona correctament.

- Les optimitzacions aplicades són incrementals, és a dir, que una optimització conté les anteriors fins a l'actual, i així successivament.

- Pel que fa als temps d'execució que apareixen al llarg d'aquest document:

  - Representen el *elapsed time* de la part instrumentada del codi, delimitada per la normativa de PCA. La mesura l'hem realitzat mitjançant la crida a sistema `gettimeofday`.

  - Per a cada optimització aplicada, el programa s'ha executat 10 vegades seguides per a cadascun dels 3 tests, calculant així la mitjana del temps d'execució i la seva desviació estàndard. Aquesta tasca l'hem automatitzat mitjançant el *script* `timing_and_stddev.sh` adjunt al comprimit de la pràctica. Per a conéixer el seu funcionament consulteu el fitxer `README.txt` també adjunt.

  - En cadascuna d'aquestes execucions s'ha redireccionat la sortida a `/dev/null`.

  - Durant la realització de les mesures dels temps, l'ordinador sobre el qual s'han dut a terme no tenia altres aplicacions obertes ni corrent en segon terme.

- Per corroborar amb fermesa la hipòtesi que una versió del programa és millor que una altra , hem realitzat la prova estadística t-student. Aquesta prova és un contrast d'hipòtesis unilateral, ja que tenim indicis que una versió sempre és millor que l'altre. Tanmateix, parteix de les premisses que les dues mostres són mostres aleatòries simple que segueixen una distribució normal, i les variàncies són desconegudes però idèntiques. Per a fer aquests prova hem recorregut a la fulla de càlcul `comparison.odt` adjunta al comprimit de la pràctica. Si no es diu el contrari, totes les proves (una per cada optimització aplicada), amb risc $\alpha = 0.05$, han donat com a resultat que es pot rebutjar la hipòtesis nul·la H_0 (els temps d'execució dels dos programes són iguals) i, per tant, podem assegurar que hi ha prou evidència com per creure que la versió optimitzada és millor que la seva antecessora.

- Totes aquestes condicions tenen com a objectiu garantir una obtenció rigorosa i fiable de les mesures, amb el mínim de soroll possible per a no distorsionar la qualitat dels resultats i posteriors anàlisis.

- Dins el comprimit de la pràctica, en el directori `codis_optimitzats`, s'adjunten també els codis font optimitzats. Per saber com instal·lar-los per tal d'executar el programa amb aquesta nova versió consulteu el fitxer `README.txt`.

2. Procés d'optimització del programa

2.1 Tria de les millors opcions del compilador

Abans de començar a modificar codi per aplicar tècniques d'optimització, hem analitzat el programa original amb l'eina de *profiling* `gprof`. Els resultats han estat els següents (només es mostren les 5 rutines que més recursos del processador consumeixen dins del programa):

```
Each sample counts as 0.01 seconds.
 %   cumulative   self              self     total
time   seconds   seconds  calls  us/call  us/call     name
49.11  31.06     31.06                               electric_field
30.93  50.62     19.56  1300937139 0.02   0.02       pythagoras
 1.71  51.70      1.08                               __profile_frequency
 1.61  52.72      1.02                               fftw_hc2hc_forward_generic
 1.55  53.70      0.98                               writev
```

Les dues rutines que encapçalen el llistat (`electric_field` i `pythagoras`) representen quasi el 80% del cost del programa, motiu pel qual ens centrarem primordialment en aplicar millores sobre aquestes dues. Si ara analitzem els seus codis podem veure com es realitzen nombroses operacions en coma flotant, de llarga latència en general. Una opció per a millorar el rendiment del programa és buscar en el manual de gcc si hi ha cap opció que agilitzi càlculs en coma flotant. Hem trobat les opcions `-mfpmath=sse, -ffast-math` i `-fno-math-errno` que, sense entrar en detall, sacrifiquen precisió en els resultats a costa de millorar el rendiment en els càlculs (consultar el manual del `gcc` per a més informació).

Una vegada escollides, les hem afegit al fitxer `Makefile`, i seguidament hem compilat el programa amb aquestes. Cal notar que aquestes opcions no s'habiliten per defecte quan activem l'opció `-O3`. Si ara tornem a fer *profiling* amb `gprof` podem apreciar com s'han disminuït els temps consumits per a les funcions `electric_field` i `pythagoras`:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self total
 time   seconds   seconds calls  us/call  us/call  name
 46.95 25.42    25.42                             electric_field
 30.27 41.81    16.39 1300937139     0.01 0.01        pythagoras
  1.98 42.88    1.07                               fftw_hc2hc_forward_generic
  1.87 43.90    1.02                              fftw_no_twiddle_11
  1.72 44.83    0.93                              writev
```

Els temps, desviacions tipus i *speedups* obtinguts després d'executar el banc de proves són els següents (Taula 1):

	Versió amb opcions del gcc		Versió prèvia (original)		*Speedup*	
	Temps (segons)	Desviació estàndard	Temps (segons)	Desviació estàndard	Respecte la versió prèvia	Respecte la versió original
Test 1	39.992225	0.101335	49.487280	0.143446	**1,24X**	**1,24X**
Test 2	153.299602	0.667342	188.698195	0.238435	**1,23X**	**1,23X**
Test 3	140.619830	0.259407	177.827360	0.220456	**1,26X**	**1,26X**

Taula 1: Resultats del banc de proves del programa compilat amb les opcions matemàtiques del gcc respecte la versió original.

2.2 Primera optimització

Si centrem la nostra atenció a la sortida generada pel `gprof` en l'actual versió del programa:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self total
 time   seconds   seconds calls  us/call  us/call  name
 46.95     25.42 25.42                             electric_field
 30.27     41.81 16.39 1300937139     0.01 0.01  pythagoras
  1.98     42.88     1.07                               fftw_hc2hc_forward_generic
  1.87     43.90     1.02                              fftw_no_twiddle_11
  1.72     44.83     0.93                              writev
```

Les rutines que consumeixen més recursos de CPU continuen sent `electric_field` i `pythagoras`. Si tornem a analitzar els seus codis, ens adonem que la funció `pythagoras` es composa únicament d'una línia, i segons `gprof` es crida un nombre considerable de vegades. Una bona opció per a reduir el cost que suposen les crides a rutines dins d'un programa és aplicar *inlining*. Mitjançant una macro declarada al fitxer `electrostatics.c` hem aconseguit implementar aquesta millora. A continuació es mostra com hem definit aquesta macro:

```
#define PYTHAGORAS(x1,y1,z1,x2,y2,z2) sqrt(((x1-x2)*(x1-x2))+((y1-y2)*(y1-
y2))+((z1-z2)*(z1-z2)));
```

Per altra banda, si ara centrem l'atenció en el codi de la rutina `electric_field`, a simple vista no podem apreciar quines parts o línies de codi són les que consumeixen més. Per això tornem a fer *profiling* amb `gprof` amb una granularitat de línia de codi. Obtenim els següents resultats (només mostrem aquelles línies de codi de la funció `electric_field` que representen més d'un 3% del cost total del programa):

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self  total
 time   seconds   seconds    calls  ns/call  ns/call  name
 13.73  7.51      7.51                      electric_field (electrostatics.c:163 @ 804c770)
 13.64 14.97      7.46                      pythagoras (coordinates.c:51 @ 804c234)
  8.98 19.88      4.91                      pythagoras (coordinates.c:53 @ 804c251)
  6.10 23.22      3.34                      electric_field (electrostatics.c:135 @ 804c6a0)
  5.92 26.46      3.24                      electric_field (electrostatics.c:139 @ 804c6d9)
  5.80 29.63      3.18                      electric_field (electrostatics.c:139 @ 804c71d)
  4.73 32.22      2.59                      electric_field (electrostatics.c:134 @ 804c79c)
  4.58 34.73      2.51                      electric_field (electrostatics.c:137 @ 804c6ce)
  4.31 37.09      2.36                      pythagoras (coordinates.c:51 @ 804c225)
  3.63 39.07      1.99                      electric_field (electrostatics.c:143 @ 804c72d)
```

La línia més costosa, amb diferència, és la línia 163, que conté el càlcul de la variable `phi`. Aquesta línia fa servir operacions de llarga latència i, a simple vista, no veiem cap manera d'optimitzar els càlculs. La resta de línies que més consumeixen dins de la funció fan referència a branques condicionals. Aquest és l'extracte del codi font que representa el conjunt d'aquestes línies:

```
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 ;
    if( distance >= 2.0 ) {
        if( distance >= 8.0 ) {
            epsilon = 80 ;
        } else {
              if( distance <= 6.0 ) {
                  epsilon = 4 ;
              } else {
                    epsilon = ( 38 * distance ) - 224 ;
              }
        }
        phi += ( This_Structure.Residue[residue].Atom[atom].charge / ( epsilon *
distance ) ) ;
    }
}
```

Totes aquestes branques es poden reescriure d'una manera més òptima i senzilla d'entendre, tant pel compilador com per al propi programador. El propòsit és el de reduir el cost que suposen els salts condicionals i les possibles penalitzacions en cas de predicció errònia.

Després d'haver reestructurat el codi de diverses maneres (aplicant mecanismes de *bithacks*, intercanviant l'ordre de les condicions, eliminant branques redundants, etc.), el codi reestructurat que proporciona un millor rendiment és el següent:

```
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 < 8) {
        if( distance <= 6.0 ) {
            epsilon = 4 ;
            if( distance < 2.0 ) distance = 2.0 ;
        } else {
              epsilon = ( 38 * distance ) - 224 ;
        }
         phi += ( This_Structure.Residue[residue].Atom[atom].charge / ( epsilon *
distance ) ) ;
    }
}
```

Si ara tornem a fer *profiling* amb `gprof` a nivell de línia de codi podem comprovar com hem disminuït el cost que suposen aquestes línies respecte la versió anterior:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self  total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 31.20  9.11      9.11                        electric_field (electrostatics.c:141 @ 804c6ca)
 11.13 12.36      3.25                        electric_field (electrostatics.c:139 @ 804c6c0)
  6.04 14.13      1.77                        electric_field (electrostatics.c:137 @ 804c73d)
  5.72 15.80      1.67                        electric_field (electrostatics.c:136 @ 804c74b)
  3.89 16.93      1.14                        electric_field (electrostatics.c:157 @ 804c731)
  3.66 18.00      1.07                        fftw_hc2hc_forward_generic
  3.25 18.95      0.95                        electric_field (electrostatics.c:149 @ 804c6b3)
  3.13 19.87      0.92                        fftw_no_twiddle_11
  3.07 20.76      0.90                        fftw
  2.76 21.57      0.81                        fftw_hc2hc_backward_generic
  2.33 22.25      0.68                        fftw_twiddle_generic
  2.09 22.86      0.61                        electric_field (electrostatics.c:145 @ 804c707)
  1.95 23.43      0.57                        fftw_executor_simple
  1.73 23.93      0.51                        fftw_no_twiddle_10
  1.58 24.39      0.46                        fftw_twiddle_rader
  1.54 24.84      0.45                        fftw_no_twiddle_5
  1.49 25.28      0.44                        electric_field (electrostatics.c:137 @ 804c6a0)
```

Si generem el *profiling* amb `gprof` a nivell de crida a funcions obtenim els següents resultats:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self  total
 time   seconds   seconds    calls  ns/call  ns/call  name
 65.31      19.07 19.07                             electric_field
  3.66      20.14       1.07                       fftw_hc2hc_forward_generic
  3.13      21.06       0.92                       fftw_no_twiddle_11
  3.07      21.95       0.90                       fftw
  2.76      22.76       0.81                       fftw_hc2hc_backward_generic
  2.33      23.44       0.68                       fftw_twiddle_generic
```

Podem comprovar com ara no es fa cap crida a la rutina `pythagoras` gràcies a la tècnica de *inlining* aplicada.
Una vegada hem vist que les optimitzacions han millorat aquells aspectes que ens havíem proposat millorar, executem la nova versió per obtenir la mesura dels temps d'execució (Taula 2):

	Versió amb optimització 1		Versió prèvia		*Speedup*	
	Temps (segons)	Desviació estàndard	Temps (segons)	Desviació estàndard	Respecte la versió prèvia	Respecte la versió original
Test 1	33.412458	0.047238	39.992225	0.101335	**1,20X**	**1,48X**
Test 2	126.965155	0.175984	153.299602	0.667342	**1,21X**	**1,49X**
Test 3	113.728949	0.170887	140.619830	0.259407	**1,24X**	**1,56X**

Taula 2: Resultats del banc de proves del programa amb la optimització 1 respecte la versió prèvia.

2.2 Segona optimització

Si fem *profiling* amb `gprof` a nivell de línia de codi per analitzar novament la versió actual del programa obtenim els següents resultats (només es mostres les 5 línies amb cost més significatiu):

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self  total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 31.20  9.11      9.11                       electric_field (electrostatics.c:141 @ 804c6ca)
 11.13 12.36      3.25                       electric_field (electrostatics.c:139 @ 804c6c0)
  6.04 14.13      1.77                       electric_field (electrostatics.c:137 @ 804c73d)
  5.72 15.80      1.67                       electric_field (electrostatics.c:136 @ 804c74b)
  3.89 16.93      1.14                       electric_field (electrostatics.c:157 @ 804c731)
```

La línia 141 del fitxer `electrostatics.c` representa el càlcul de la distància que efectua la macro `PYTHAGORAS`. Abans de començar a pensar en com accelerar aquest costós càlcul, ens centrarem en la segona línia que més consumeix de tot el programa, que és la 139 del mateix fitxer:

```
if( This_Structure.Residue[residue].Atom[atom].charge != 0 )
```

Entenent el funcionament del codi creiem que aquesta línia és costosa per dos motius: primer, perquè en accedir al camp `charge` d'aquesta estructura es deuen fer moltes falles a la memòria cau, i les penalitzacions per portar la dada desitjada d'altres nivells de la jerarquia de memòria té un pes considerable; segon, el predictor *hardware* falla més del que ens agradaria perquè els valors de `charge` no segueixen un patró deduïble pels mecanismes del processador. A més, si analitzem on està situada aquesta línia podem veure que està dins de 5 bucles, els 3 bucles més externs recorren un pla en 3 dimensions, i els altres 2 més interns recorren, per a cada punt del pla, tots els àtoms de l'estructura `This_Structure`. Per tant, estem accedint al camp `charge` múltiples vegades, i per aquest motiu el cost d'aquesta línia té un pes considerable Abans de començar a aplicar millores fem *profiling* del programa amb l'eina `Oprofile`, per a obtenir el nombre d'accessos totals i el nombre de falles que es realitzen a la memòria cau L1. Els resultats (Taula 3) demostren uns nombre molt elevats d'accessos i falles.

Per a millorar aquesta part del programa crearem un vector amb la informació indispensable per a efectuar els càlculs de dins la branca condicional. Aquesta informació indispensable són els camps `charge` i les tres coordenades dels àtoms. És a dir, un vector d'estructures, on cada element serà una estructura de 4 camps que representaran la càrrega i les 3 coordenades dels àtoms. Aquest estructura la declarem al fitxer `structures.h`, i té el següent format:

```
struct Atom_Charged{
    float       charge ;
    float       coord[3] ;
} ;
```

Tanmateix, com que només fem càlculs sobre aquells àtoms que tenen càrrega no neutra, en aquest vector només ens guardarem els que satisfan aquesta condició. D'aquesta manera estalviem memòria, ja que el vector no haurà de ser de tants elements, i per altra banda, com que en aquest vector només guardarem els àtoms amb càrrega no neutra no serà necessari pagar la comprovació de la condició, eliminant d'aquesta manera la branca.

Després d'estudiar amb atenció com es van cridant les diferents funcions d'inicialització de les estructures en el programa principal, hem decidit que el millor lloc on inicialitzar el nostre nou vector d'estructures és dins del fitxer font `manipulate_structures.c`, dins de la funció `translate_structure_onto_origin`. Com que només volem afegir aquesta inicialització en el cas de la proteïna estàtica, fem una especialització de la funció original. La nova funció l'hem anomenat té la següent capçalera (afegida al fitxer `structures.h`):

```
extern struct Structure translate_static_structure_onto_origin( struct
Structure This_Structure, int num_charged_atoms, struct Atom_Charged
*charged_atoms )
```

La inicialització l'efectuem una vegada s'han assignat les coordenades a cada àtom, això és, just al final de la funció. L'extracte del codi de la inicialització és el següent:

```
/* Translate */

  int k = 0;
  for( residue = 1 ; residue <= New_Structure.length ; residue ++ ) {

      for( atom = 1 ; atom <= New_Structure.Residue[residue].size ; atom ++ ) {

          New_Structure.Residue[residue].Atom[atom].coord[1] -= average_x ;
          New_Structure.Residue[residue].Atom[atom].coord[2] -= average_y ;
          New_Structure.Residue[residue].Atom[atom].coord[3] -= average_z ;

          if (New_Structure.Residue[residue].Atom[atom].charge != 0) {
              charged_atoms[k].charge = New_Structure.Residue[residue].Atom[atom].charge;
              charged_atoms[k].coord[0] = New_Structure.Residue[residue].Atom[atom].coord[1] ;
              charged_atoms[k].coord[1] = New_Structure.Residue[residue].Atom[atom].coord[2] ;
              charged_atoms[k].coord[2] = New_Structure.Residue[residue].Atom[atom].coord[3] ;
              ++k;
          }
      }
  }
```

Com que a priori no coneixem el nombre d'àtoms amb càrrega no neutre, primer hem de comptar-los, i aleshores demanar memòria per al nostre vector `charged_atoms`. Aquest comptatge l'efectuem a la funció `assign_charges`, del fitxer font `electrostatics.c`. Com que només volem comptar els àtoms carregats de la proteïna estàtica, també apliquem una especialització de la funció com hem fet prèviament. La capçalera de la nova funció, afegida al fitxer `structures.h`, és la següent:

```
extern void assign_charges_static( struct Structure This_Structure, int
*num_charged_atoms ) ;
```

El codi que fa el comptatge l'hem col·locat al final del bucle més intern de la funció, un cop la càrrega s'ha assignat a cada àtom. L'extracte es mostra a continuació:

```
if (This_Structure.Residue[residue].Atom[atom].charge != 0) {
    *num_charged_atoms += 1;
```

I aleshores demanem la memòria necessària per aquest vector, dins del fitxer font `ftdock.c`, just després d'assignar les càrregues als àtoms de les proteïnes. L'extracte del codi és el següent:

```
  /* Assign charges */

  if( electrostatics == 1 ) {
     printf( "Assigning charges\n" ) ;
     assign_charges_static( Static_Structure, &num_charged_atoms ) ;
     assign_charges( Mobile_Structure ) ;
  }
```

```
  posix_memalign((void **)&charged_atoms, 16, num_charged_atoms *
sizeof_Atom_Charged);
```

Hem fet servir la crida a `posix_memalign` i hem alineat l'adreça inicial a 16 bytes per a possibles optimitzacions futures relacionades amb tècniques de vectorització de codi.

Aleshores, una vegada implementada la millora, tornem a fer *profiling* amb `Oprofile` per a verificar que, amb aquesta millora, hem reduït el nombre d'accessos i falles a memòria cau de L1. Els resultats es mostren a continuació (Taula 3):

Versió amb optimització 2			Versió prèvia			Millora		
Accessos a L1	Falles a L1	Falles per accés	Accessos a L1	Falles a L1	Falles per accés	Accessos a L1	Falles a L1	Falles per accés
106082 * 50000	187 * 5000	0.000176	366808 * 50000	680409 * 5000	0.185495	**541.40%**	**363755.08%**	**105128.01%**

Taula 3: Resultats obtinguts amb `Oprofile` dels accessos i falles a la memòria cau L1 del programa amb la optimització 2 respecte la versió prèvia. Els accessos i falles es representen com el nombre d'esdeveniments succeïts multiplicat per la seva freqüència de mostreig.

Queda palesa que la millora és espectacular, reduint el nombre d'accessos, el nombre de falles i els ràtios de falles per accés a memòria cau L1. Els resultats dels temps d'execució una vegada hem executat el banc de proves són els següents (Taula 4):

	Versió amb optimització 2		Versió prèvia		*Speedup*	
	Temps (segons)	Desviació estàndard	Temps (segons)	Desviació estàndard	Respecte la versió prèvia	Respecte la versió original
Test 1	13.766615	0.042692	33.412458	0.047238	**2,43X**	**3,6X**
Test 2	49.985143	0.236251	126.965155	0.175984	**2,54X**	**3,78X**
Test 3	33.583494	0.119356	113.728949	0.170887	**3,39X**	**5,3X**

Taula 4: Resultats del banc de proves del programa amb la optimització 2 respecte la versió prèvia.

2.3 Tercera optimització

Arribats en aquest punt tenim la següent sortida del `gprof`:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self  total
time   seconds   seconds      calls  Ts/call  Ts/call  name
23.21      3.61  3.61                             electric_field (electrostatics.c:189 @ 804caf0)
6.83       4.67  1.06                             fftw_hc2hc_forward_generic
5.76       5.56  0.90                             fftw_no_twiddle_11
4.99       6.34  0.78                             fftw_hc2hc_backward_generic
4.89       7.10  0.76                             fftw
4.64       7.82  0.72                             fftw_twiddle_generic
3.83       8.41  0.60                             fftw_no_twiddle_10
```

La crida a `PYTHAGORAS` segueix sent el coll d'ampolla de la nostra aplicació. Per tal d'eliminar-lo s'ha intentat canviar el punt de vista del problema i aplicar optimitzacions al nivell de l'algorisme. Hem descobert que la funció `electric_field` fa un escaneig d'un espai 3D i en funció de cada punt (x, y, z) es calcula la distància respecte cada àtom. Si aquesta és menor que 8 s'acumula en una posició de memòria, corresponent al centre (x, y, z), un valor que depen de la distància i de la càrrega de l'àtom.

Si girem el problema fixant un àtom i variant els centres ens hem donat que es poden aplicar optimitzacions que potencialment podrien reduir significativament el nombre de crides a la macro `PYTHAGORAS`. A continuació es fa una descripció de la nova forma de veure el problema.

Considerem una esfera de radi 8 i centre (x', y', z') que és la coordenada de l'àtom. Donat un punt (x, y, z) que obtenim mitjançant l'escaneig del mapa 3D no cal calcular Pitàgores en 3 dimensions si `abs(x' - x) > 8` o `abs(y' - y) > 8` o `abs(z' - z) > 8`. En les condicions anteriors és evident que Pitàgores donarà un valor superior a 8. Geomètricament, això vol dir calcular Pitàgores només pels punts del mapa que pertanyen al cub dins del qual està circumscrita la esfera que havíem mencionat.

Abans de començar a implementar aquesta millora vam fer unes proves per veure en quantes iteracions del bucle més intern s'entrava en el cos de la branca condicional que comprova si la distància és més petita que 8. Geomètricament volíem saber, en total, quantes vegades es calcula Pitàgores per punts externs al cub de cada àtom. A la Taula 5 es poden veure aquests resultats, contrastats amb els resultats que dóna la versió del programa amb la tercera optimització.

A continuació mostrem el codi resultant de la funció `electric_field` un cop implementada la idea descrita anteriorment:

```
for( atoms = 0; atoms < num_charged_atoms; atoms ++ ) {
  for( x = 0 ; x < grid_size ; x ++ ) {
    printf( "." ) ;
    x_centre  = gcentre( x , grid_span , grid_size ) ;
    if (abs(charged_atoms[atoms].coord[0] - x_centre) < 8.0) {
      for( y = 0 ; y < grid_size ; y ++ ) {
        y_centre  = gcentre( y , grid_span , grid_size ) ;
        if (abs(charged_atoms[atoms].coord[1] - y_centre) < 8.0) {
          for( z = 0 ; z < grid_size ; z ++ ) {
            z_centre  = gcentre( z , grid_span , grid_size ) ;
             if (abs(charged_atoms[atoms].coord[2] - z_centre) < 8.0) {
               distance = PYTHAGORAS( charged_atoms[atoms].coord[0] ,
charged_atoms[atoms].coord[1] , charged_atoms[atoms].coord[2] , x_centre , y_centre ,
z_centre ) ;
              if (distance < 8) {
                if( distance <= 6.0 ) {
                  if( distance < 2.0 ) distance = 2.0 ;
                  epsilon = 4 ;
                } else {
                    epsilon = ( 38 * distance ) - 224 ;
                }
                grid[gaddress(x,y,z,grid_size)] += (fftw_real)
( charged_atoms[atoms].charge / ( epsilon * distance ) ) ;
            }
//S'ometen les claus per a una major llegibilitat
```

Podem veure, per exemple, que només en detectar que la variable `x_centre` corresponent a una determinada iteració de les x no es faran les `grid_size^2 * num_charged_atoms` iteracions que fèiem abans. No obstant, ara perdem localitat amb l'escriptura al vector 3D `grid` ja que amb la versió prèvia del programa es feia servir un acumulador i `grid` s'actualitzava només una vegada després d'haver analitzat tots els àtoms per un determinat centre. Però vistos els resultats dels rendiments (Taula 6), val la pena sacrificar aquesta pèrdua de localitat.

Si ara tornem a fer *profiling* amb `gprof`, el cost de la funció `electric_field` ha baixat tant que té un pes inferior a moltes de les funcions que apliquen transformacions de *Fourier*:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self  total
time   seconds   seconds    calls  ns/call  ns/call  name
  9.83      1.06  1.06                             fftw_no_twiddle_11
  9.65      2.10  1.04                             fftw_hc2hc_forward_generic
  8.49      3.02  0.92                             fftw
  7.28      3.80  0.79                             fftw_hc2hc_backward_generic
  6.86      4.54  0.74                             fftw_twiddle_generic
  5.33      5.12  0.58                             fftw_no_twiddle_5
  4.55      5.61  0.49                             fftw_no_twiddle_10
  3.99      6.04  0.43                             fftw_executor_simple
  3.06      6.37  0.33                             fftw_no_twiddle_2
  2.69      6.66  0.29                             __udivmoddi4 (libgcc2.c:881 @ 8096201)
  2.69      6.95  0.29                             fftwi_twiddle_generic
  2.69      7.24  0.29                             executor_many
  2.60      7.52  0.28                             fftw_twiddle_6
  2.60      7.80  0.28                             fftw_twiddle_rader
  2.46      8.06  0.27                             electric_field (electrostatics.c:191 @
804cc41)
  2.27      8.31  0.25                             electric_field (electrostatics.c:189 @ 804cc27)
```

A continuació es mostra la Taula 5 descrita en paràgrafs anteriors, i la Taula 6, que mostra els resultats dels temps d'execució un cop executat el banc de proves a la nova versió del programa:

	Versió amb optimització 3		Versió prèvia		Millora	
	distance < 8	distance >= 8	distance < 8	distance >= 8	distance < 8	distance >= 8
Test 1	3416177	6529585	3416177	1299296160	**1**	**19798.60%**
Test 2	8305165	15884259	8305165	746673152	**1**	**4600.71%**
Test 3	8153571	15576310	8153571	927301792	**1**	**5853.28%**

Taula 5: Resultats obtinguts, mitjançant comptadors, del nombre de vegades que el programa entra dins de la branca condicional `if (distance < 8)`, i quantes no hi entra, entre la versió optimitzada i l'anterior.

	Versió amb optimització 3		Versió prèvia		*Speedup*	
	Temps (segons)	Desviació estàndard	Temps (segons)	Desviació estàndard	Respecte la versió prèvia	Respecte la versió original
Test 1	9.033557	0.041247	13.766615	0.042692	**1,52X**	**5,48X**
Test 2	31.398443	0.320075	49.985143	0.236251	**1,59X**	**6,01X**
Test 3	14.053634	0.113620	33.583494	0.119356	**2,39X**	**12,65X**

Taula 6: Resultats del banc de proves del programa amb la optimització 3 respecte la versió prèvia.

La destacada reducció en el nombre de vegades que es crida a `PYTHAGORAS` (Taula 5) ha resultat ser determinant en la consecució dels rendiments mostrats en la Taula 6. Noteu que el nombre de vegades que s'entra a la branca condicional de si la distància és menor que 8 és el mateix tant en la versió prèvia com en la versió amb l'optimització 3, demostrant com aquesta nova millora no altera la correctesa de l'algorisme del programa original.

2.4 Quarta optimització

Considerant les actuals dades obtingudes amb l'eina `gprof`:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self  total
time   seconds   seconds     calls  ns/call  ns/call  name
  9.83      1.06  1.06                          fftw_no_twiddle_11
  9.65      2.10  1.04                          fftw_hc2hc_forward_generic

  ...       ...   ...                           ...        ...         ...

  2.60      7.80  0.28                          fftw_twiddle_rader
  2.46      8.06  0.27                          electric_field (electrostatics.c:191 @
804cc41)
  2.27      8.31  0.25                          electric_field (electrostatics.c:189 @ 804cc27)
```

encara podem aplicar una nova millora a la funció `electric_field`. Si atenem a la línia 189 veiem com el seu cost ve determinat per la crida a la macro `gcentre`:

```
z_centre  = gcentre( z , grid_span , grid_size ) ;
```

La declaració d'aquesta macro es troba en el fitxer font `structures.h`:

```
#define gcentre(ordinate,grid_span,grid_size) ( (float)(ordinate) + .5 ) *
(float)( (grid_span) / (float)(grid_size) ) - (float)( (grid_span) / 2 )
```

Ens hem donat compte que `g_centre` té dos paràmetres que són sempre constants i que varia linealment en funció del primer, que és la variable de control. Com es tracta d'una funció lineal, podem precalcular m i n de la equació de la recta on "y = m*x + n" és la expressió. La pendent seria `(float)( (grid_span) / (float)(grid_size) ) - (float)( (grid_span) / 2 )` (ja que multiplica a la variable ordinate), i la n seria `0.5 * (float)( (grid_span) / (float)(grid_size) ) - (float)( (grid_span) / 2 )`.

Cal fer aquests dos càlculs només una vegada ja que per tota la funció `electric_field` aquesta macro es crida amb paràmetres `grid_span` i `grid_size` fixes. Per tant, cal afegir aquestes línies de codi per al càlcul previ, dins de la funció `electric_field` (dins del fitxer font `electrostatics.c`), abans de començar el recorregut dels bucles:

```
float slope = (float)( (grid_span) / (float)(grid_size) );
float n = 0.5 * slope - (float)( (grid_span) / 2 );
for( atoms = 0; atoms < num_charged_atoms; atoms ++ ) {
...
```

i definir una nova macro al fitxer font `structures.h`, que serà la que cridarem en el nou codi, amb els dos paràmetres fixes `slope` i `n` i la variable d'iteració `ordinate`:

```
#define gcentre_opt(ordinate,slope,n) ( (float)(ordinate) ) * slope + n
```

Observem com almenys s'ha eliminat una divisió que és una operació de llarga latència (suposem que la divisió per 2 és optimitzada d'alguna forma pel compilador encara que sigui una operació en coma flotant).

Si ara fem novament *profiling* amb `gprof` observem com hem reduït el pes que suposa aquesta línia, que és ínfim:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 10.25      1.08     1.08                             fftw_no_twiddle_11
 10.06      2.14     1.06                             fftw_hc2hc_forward_generic
  ...       ...      ...                              ...       ...       ...
  0.57      9.70     0.06                             electric_field
(electrostatics.c:192 @ 804cbee) (electrostatics.c:194 @ 804cbff)
```

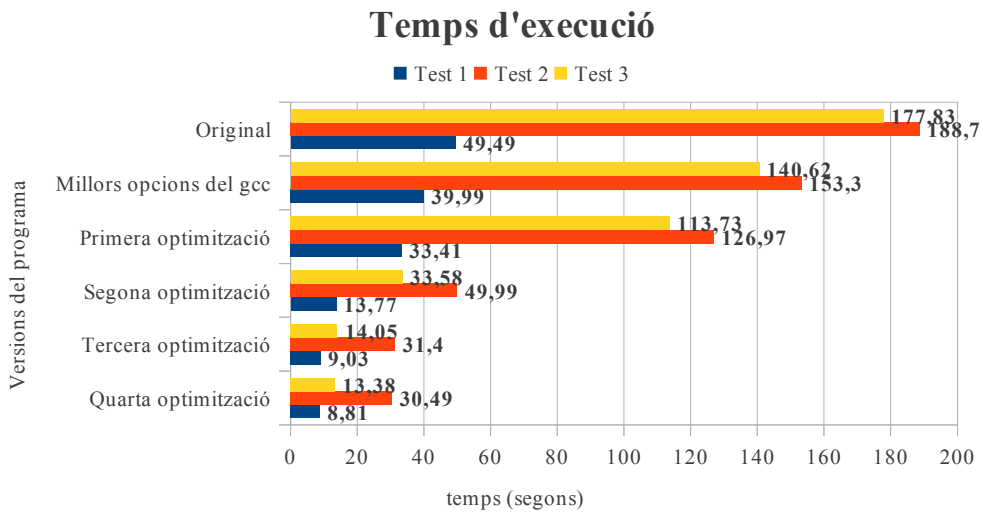
Una vegada hem executat la nova versió amb el banc de proves, obtenim els següents resultats en els temps d'execució (Taula 7):

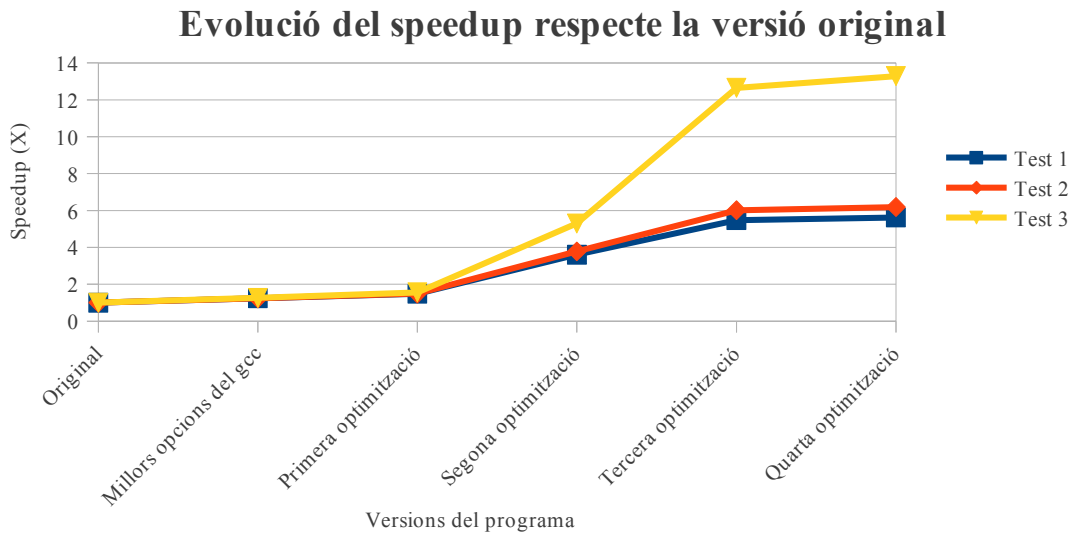
	Versió amb optimització 4		Versió prèvia		*Speedup*	
	Temps (segons)	Desviació estàndard	Temps (segons)	Desviació estàndard	Respecte la versió prèvia	Respecte la versió original
Test 1	8.809943	0.040261	9.033557	0.041247	**1,03X**	**5,62X**
Test 2	30.489260	0.269634	31.398443	0.320075	**1,03X**	**6,19X**
Test 3	13.377031	0.119644	14.053634	0.113620	**1,05X**	**13,29X**

Taula 7: Resultats del banc de proves del programa amb la optimització 4 respecte la versió prèvia.

3. Valoració final

Aquest procés incremental fins arribar a obtenir la versió més òptima possible del programa ha esdevingut tots els resultats mostrats al llarg del present document. Es pot apreciar com cada optimització a tingut un impacte més o menys notori en la millora del rendiment del programa. En termes generals, la tria de les millors opcions del compilador i la primera optimització han representat millores petites però representatives. En mitjana, aquestes millores es mouen entorn al 20-25% respecte la versió prèvia a aquestes. Aleshores tenim la segona i tercera optimització, que són les que més han accelerat el programa. Per a la segona optimització les millores se situen entorn a un 150% en els 2 primers tests, i en el tercer la millora augmenta fins al 239%. respecte la versió anterior. Pel que fa a la tercera optimització, els guanys obtinguts es situen entre els 140-150% respecte la segona optimització. I per últim, la quarta optimització, tant sols aconsegueix un 5% de millora respecte la versió amb la tercera optimització. Si fem un anàlisi global, enfrontant la versió més òptima (quarta optimització) contra l'original, obtenim uns *speedups* molt meritoris: 5,62X (millora del 462%) per al test 1, 6,19X (millora del 519%) per al test 2, i un impressionant 13,29X (millora del 1229%) per la test 3. El següents gràfic mostra una visió general de l'evolució en els temps d'execució i dels *speedups* per a totes les versions del programa i per a cadascun dels 3 tests:





4. Annexos

Annex A. Entorn d'execució

La informació següent s'ha obtingut executant la comanda `x86info`:

- Model CPU: Intel Core2 Duo E8400 3,00GHz.
- Memòria cau d'instruccions L1: 32KB, associativitat de grau 8, 64 Bytes per línia.
- Memòria cau de dades L1: 32KB, associativitat de grau 8, 64 Bytes per línia.
- Memòria cau de dades L2: 6MB, associativitat de grau 24, 64 Bytes per línia.
- TLB d'instruccions: 4x 4MB o 8x 2MB entrades, associativitat de grau 24.
- TLB d'instruccions: 4K pàgines, associativitat de grau 4, 128 entrades.
- TLB de dades: pàgines de 4MB, associativitat de grau 4, 32 entrades.
- TLB de dades L1: pàgines de 4KB, associativitat de grau 4, 16 entrades.
- TLB de dades L1: pàgines de 4MB, associativitat de grau 4, 16 entrades.
- TLB de dades: 4K pàgines, associativitat de grau 4, 256 entrades.
- 64 Bytes de *prefetching*.
- Mida de les adreces : 36 bits per a les físiques, 48 bits per a les virtuals.
- El processador disposa de 2 *threads*.

Annex B. Altres optimitzacions no aplicades

Al llarg del present document s'han presentat tota una sèrie de millores aplicades en base als anàlisis obtinguts mitjançant diferents eines de *profiling*. Però no totes les millores que hem aplicat han esdevingut en èxit, i les hem hagut de retirar perquè no aportaven cap millora en el rendiment. Aquest apartat està dedicat a explicar quines han estat aquelles optimitzacions que, en un principi, ens pensàvem que millorarien l'aplicació però, malauradament, després d'executar els bancs de prova ens vam adonar que no ho feien.

La primera optimització que no ens va donar fruits va ser una vectorització en el codi del programa en la optimització 2. Veien els resultats que donava `gpof` volíem accelerar el càlcul de `PYTHAGORAS` intentant computar en paral·lel totes les operacions que s'aplicaven abans de fer l'arrel quadrada. L'extracte del codi resultant és el següent (la versió completa es pot consultar al fitxer font `electrostatics_vec.c` dins del directori `opts_no_aplicades` adjunt al comprimit de la pràctica):

```
for( x = 0 ; x < grid_size ; x ++ ) {
   printf( "." ) ;
   x_centre  = gcentre( x , grid_span , grid_size ) ;
   for( y = 0 ; y < grid_size ; y ++ ) {
     y_centre  = gcentre( y , grid_span , grid_size ) ;
     for( z = 0 ; z < grid_size ; z ++ ) {
       z_centre  = gcentre( z , grid_span , grid_size ) ;
       phi = 0 ;
        __m128 centre = _mm_set_ps( z_centre, y_centre, x_centre, 0.0);
        for( atoms = 0; atoms < num_charged_atoms; atoms ++ ) {
          float xyz_sqrt[4];
          __m128 punt = _mm_load_ps((float *)&charged_atoms[atoms]);
          __m128 res = _mm_sub_ps(punt, centre);
          res = _mm_mul_ps(res, res);
          _mm_store_ps(&xyz_sqrt[0], res); // xyz_sqrt[0] conté basura! (no vàlid per als
                                           // càlculs)
          distance = sqrt(xyz_sqrt[1] + xyz_sqrt[2] + xyz_sqrt[3]);
          if( distance < 2.0 ) distance = 2.0 ;
          ...          ...          ...
```

Però vistos els resultats en els temps d'execució i en els posteriors anàlisis amb l'eina `gprof`, vàrem deduir que allò que realment ens penalitzava era el cost de fer l'arrel quadrada. Per aquest motiu, vam rebutjar l'anterior vectorització per aplicar una de nova que computés l'arrel quadrada per cada quatre àtoms. Per aquests propòsit vam aplicar *unrolling* de grau 4 sobre el bucle que recorre els àtoms. A més, vam redefinir la macro `PYTHAGORAS` per eliminar l'operació de l'arrel quadrada, ja que en aquest cas la faríem una sola vegada cada quatre àtoms, i de forma vectorial. L'extracte del codi resultant és el següent (la versió completa es pot consultar al fitxer font `electrostatics_vec2.c` dins del directori `opts_no_aplicades` adjunt al comprimit de la pràctica):

```
#define PYTHAGORAS(x1, y1, z1, x2, y2, z2) ( ( x1 - x2 ) * ( x1 - x2 ) ) + ( ( y1 - y2 ) * ( y1 -
y2 ) ) + ( ( z1 - z2 ) * ( z1 - z2 ) );
...          ...          ...
...          ...          ...
for( x = 0 ; x < grid_size ; x ++ ) {
   printf( "." ) ;
   x_centre  = gcentre( x , grid_span , grid_size ) ;
   for( y = 0 ; y < grid_size ; y ++ ) {
     y_centre  = gcentre( y , grid_span , grid_size ) ;
     for( z = 0 ; z < grid_size ; z ++ ) {
       z_centre  = gcentre( z , grid_span , grid_size ) ;
       phi = 0 ;
        __m128 centre = _mm_set_ps( z_centre, y_centre, x_centre, 0.0);

        for( atoms = 0; atoms < num_charged_atoms - 3; atoms+=4 ) {
          float distances[4];
          distances[0] = PYTHAGORAS(charged_atoms[atoms].coord[0], charged_atoms[atoms].coord[1],
charged_atoms[atoms].coord[2], x_centre, y_centre, z_centre);
          distances[1] = PYTHAGORAS(charged_atoms[atoms+1].coord[0],
charged_atoms[atoms+1].coord[1], charged_atoms[atoms+1].coord[2], x_centre, y_centre, z_centre);
          distances[2] = PYTHAGORAS(charged_atoms[atoms+2].coord[0],
charged_atoms[atoms+2].coord[1], charged_atoms[atoms+2].coord[2], x_centre, y_centre, z_centre);
          distances[3] = PYTHAGORAS(charged_atoms[atoms+3].coord[0],
charged_atoms[atoms+3].coord[1], charged_atoms[atoms+3].coord[2], x_centre, y_centre, z_centre);

          __m128 punts = _mm_load_ps(distances);
          _mm_store_ps(distances, _mm_sqrt_ps(punts));
          if (distances[0] < 8) {
          if( distances[0] <= 6.0 ) {
             if( distances[0] < 2.0 ) distances[0] = 2.0 ;
               epsilon = 4 ;
             } else {
              epsilon = ( 38 * distances[0] ) - 224 ;
           }

                phi += ( charged_atoms[atoms].charge / ( epsilon * distances[0] ) ) ;
```

```
        }

        if (distances[1] < 8) {
        if( distances[01 <= 6.0 ) {
           if( distances[1] < 2.0 ) distances[1] = 2.0 ;
             epsilon = 4 ;
           } else {
            epsilon = ( 38 * distances[0]1) - 224 ;
         }

              phi += ( charged_atoms[atoms].charge / ( epsilon * distances[1] ) ) ;

        }
        if (distances[2] < 8) {
        if( distances[02 <= 6.0 ) {
           if( distances[2] < 2.0 ) distances[2] = 2.0 ;
             epsilon = 4 ;
           } else {
            epsilon = ( 38 * distances[0]2) - 224 ;
         }

              phi += ( charged_atoms[atoms].charge / ( epsilon * distances[2] ) ) ;

        }
        if (distances[3] < 8) {
          if( distances[3] <= 6.0 ) {
            if( distances[3] < 2.0 ) distances[3] = 2.0 ;
              epsilon = 4 ;
           }  else {
                epsilon = ( 38 * distances[3] - 224 ;
           }

             phi += ( charged_atoms[atoms].charge / ( epsilon * distances[0] ) ) ;

          }

      }

      grid[gaddress(x,y,z,grid_size)] = (fftw_real)phi ;

    }
  }
}
```

Tanmateix, aquesta nova implementació tampoc ens aportava cap millora en els temps d'execució, i la vam haver de rebutjar. Va ser aleshores quan vam dedicar tots els nostres esforços en veure per quin motiu el càlcul de Pitàgores era tant costós, i entendre bé l'algorisme de la funció `electric_field`. D'aquests esforços va esdevenir l'optimització 3 explicada en el present document.

La darrere optimització que vam aplicar i que no va donar resultats de millora va ser un cop teníem el codi del programa amb la optimització 4. Vam pensar que una manera de millorar el rendiment del programa seria eliminar una sèrie d'iteracions redundants que no aportaven cap informació. Per arribar a aquesta conclusió ens vam basar en com es van calculant els punts en el pla, i ens vam adonar que, donat un àtom, per a tots els punts del pla que es recorren, si ens fixem amb la coordenada z (bucle més intern), una vegada s'han fet tots els càlculs amb aquells punts a distància menor que 8 respecte l'àtom, després tota la resta de punts seguint l'eix de les z ja seran sempre majors que 8 i que, per tant, no cal fer aquestes iteracions. Aquesta optimització anava a ser la nostra cinquena optimització, i l'extracte de les parts més rellevants del seu codi es mostra a continuació (la versió completa es pot consultar al fitxer font `electrostatics_opt5.c` dins del directori `opts_no_aplicades` adjunt al comprimit de la pràctica) :

```
float slope = (float)( (grid_span) / (float)(grid_size) );
float n = 0.5 * slope - (float)( (grid_span) / 2 );
for( atoms = 0; atoms < num_charged_atoms; atoms ++ ) {
  for( x = 0 ; x < grid_size ; x ++ ) {
   printf( "." ) ;
   x_centre  = gcentre_opt( x , slope, n ) ;
   if (abs(charged_atoms[atoms].coord[0] - x_centre) < 8.0) {
      for( y = 0 ; y < grid_size ; y ++ ) {
        y_centre  = gcentre_opt( y , slope, n ) ;
        if (abs(charged_atoms[atoms].coord[1] - y_centre) < 8.0) {
          int dist_lt8 = 0;
          int dist_lt8_prev = 0;
          for( z = 0 ; ~(dist_lt8_prev & dist_lt8) & (z < grid_size) ; z ++ ) {
            dist_lt8_prev = dist_lt8;
            z_centre  = gcentre_opt( z , slope, n ) ;
            if (abs(charged_atoms[atoms].coord[2] - z_centre) < 8.0) {
              distance = PYTHAGORAS( charged_atoms[atoms].coord[0] ,
charged_atoms[atoms].coord[1] , charged_atoms[atoms].coord[2] , x_centre , y_centre ,
z_centre ) ;
              if (distance < 8) {
                dist_lt8 = 1;
                if( distance <= 6.0 ) {
                  if( distance < 2.0 ) distance = 2.0 ;
                    epsilon = 4 ;
                  } else {
                   epsilon = ( 38 * distance ) - 224 ;
                  }
                  grid[gaddress(x,y,z,grid_size)] += (fftw_real)( charged_atoms[atoms].charge / (
epsilon * distance ) ) ;
                  }
                  else dist_lt8 = 0;
            }
            }
        }
      }
      }
    }
  }
```

Després d'aplicar el banc de proves per a observar els temps d'execució, el contrast d'hipòtesis va determinar que no teníem suficient evidència estadística com per rebutjar la hipòtesis nul·la (les dues versions triguen el mateix). I aquest factor va determinar la desestimació d'aquesta optimització.