Paral·lelisme

Lab 1: Experimental setup and tools

Grup: par2206
04/03/2020
Curs 2019-2020

Sergi Doce
Ignasi Sant

Índex

Sessió 1

Node architecture and memory

Per tal d'obtenir la informació del hardware usat a cada node, usem les comandes **lscpu** i **lstopo.** Apliquem la comanda al node boada-1, que es la usada per l'usuari, pero per als altres nodes podem modificar l'arxiu submit-*.sh per obtenir els resultats.

Obtenim la informació de cada un dels diferents nodes de Boada:

	Boada-1 to boada-4	Boada-5	Boada-6 to boada-8
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2395Mhz	2600Mhz	1700Mhz
L1-I cache size (per-core)	32KB	32KB	32KB
L1-D cache size (per-core)	32KB	32KB	32KB
L2 cache size (per-core)	256KB	256KB	256KB
Last-level cache size (per-socket)	12MB	12MB	12MB
Main memory size (per socket)	12GB	31GB	16GB
Main memory size (per node)	23GB	63GB	61GB

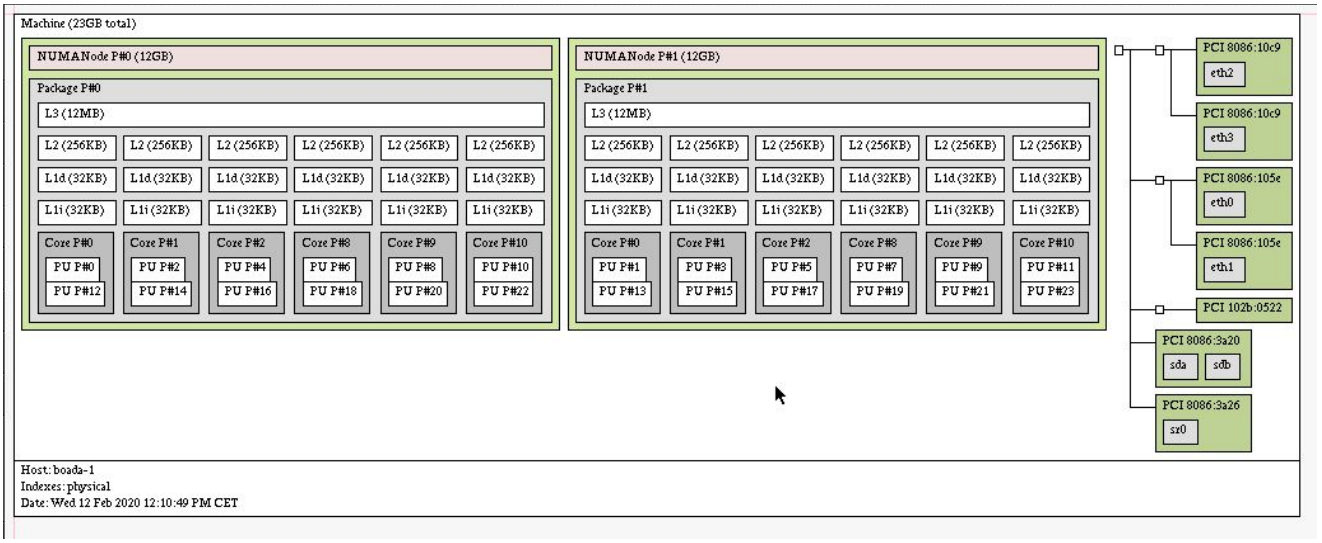


Diagrama creat usant **lstopo –of fig map.fig**

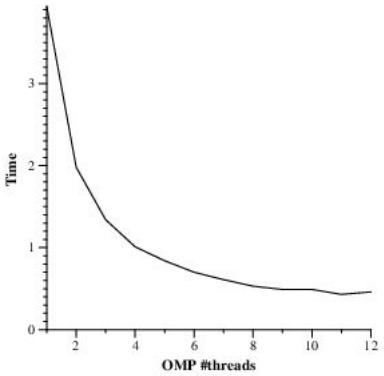
Strong vs. weak scalability

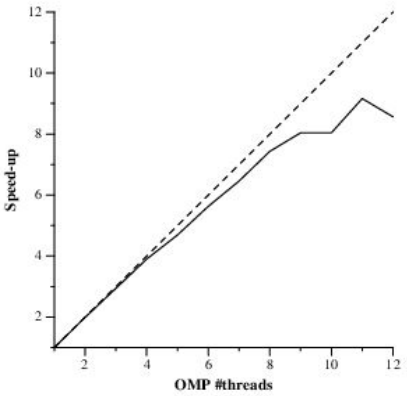
Per començar, podem definir la scalability en clau de computació com a la capacitat dels sistemes d'augmentar la seva capacitat computacional quan augmenten els recursos donats. Dit això, podem definir dues nocions dins de la escalabilitat computacional:

Strong scalability: En aquest tipus de escalabilitat es mesura com el temps d'execució canvia en funció del numero de processadors que s'usen per a un problema de tamany fixa.

Weak scalability: En aquest tipus de escalabilitat, en canvi, es mesura com el temps d'execució varia en funció del número de processadors per a un problema amb tamany fixa per processador.

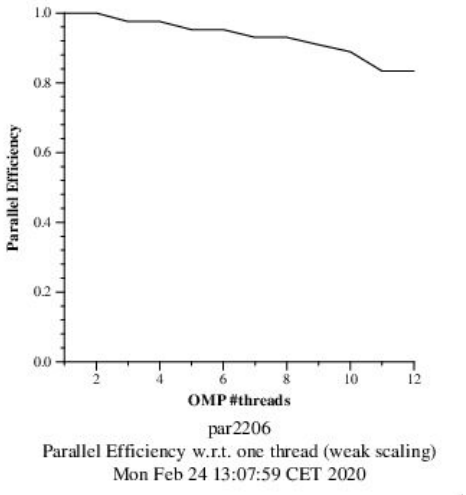
Per completar la nostra explicació farem ús de gràfics que representen el temps en funció del nombre de processadors per a un cert programa. Els gràfics corresponen al Boada 1.





Aquests dos, corresponen al gràfic de temps buscant el strong scaling i al gràfic del speed-up respectivament. Tal com hem explicat anteriorment, podem observar que el temps d'execució es redueix al aumentar el número de processadors. També veiem, però, que el temps s'estabilitza passat el processador número 10 i comença a augmentar el temps un altre cop. Això és degut al fenomen que en paral·lelisme computacional coneixem com a overhead. El overhead correspon al cost addicional per tal de poder executar un programa en paral·lel. Com veiem, arriba un punt ón l'overhead es tan alt que no val la pena seguir augmentant els recursos.

En el cas del speed-up passa una cosa semblant. No només no creix de forma totalment lineal, si no que a més cada cop decreix d'una forma més pronunciada.



Aquest últim gràfic, correspon a l'evolució de l'eficiència del paral·lelisme en el weak scaling. Com podem observar, també es va reduint poc a poc degut al overhead.

Sessió 2

Analysis of task decompositions for 3DFFT

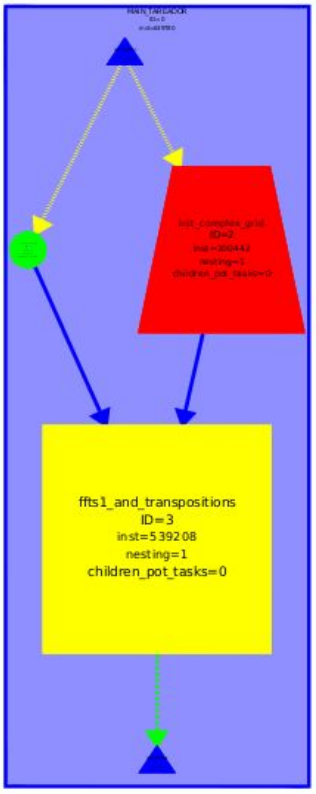
En aquesta part, analitzarem el potencial paral·lelisme que podem obtenir en un programa substituint les grans tasques en tasques cada cop més petites. Per fer això usarem el codi donat en 3dfft_tar.c, el programa Tareador i les funcions de Paraver. Hem completat la taula següent simulant la execució del programa en 1 processador i en 128 processadors respectivament, ja que era el màxim que se'ns permetia.

Version	T_1	T_∞	Parallelism
seq	639.780 ms	639.707 ms	1.000114
v1	639.780 ms	639.707 ms	1.000114
v2	639.780 ms	361.190 ms	1.771311
v3	639.780 ms	154.354 ms	4.144887
v4	639.780 ms	64.018 ms	9.993751
v5	639.780 ms	55.820 ms	11.46148

Versió seqüencial

En primer lloc, executem la versió base del programa que ens donen. Veiem que el temps en 1 processador és de 639.780 ms. Aquesta xifra es repetirà en totes les versions ja que per molt que paral·lelitzem el codi, si l'executem amb només 1 processador, no s'aprofitaran les millores.

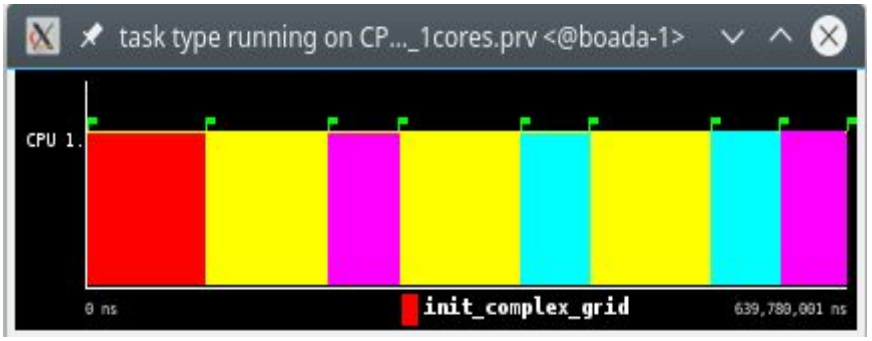
Dit això, podem veure què es produeix una lleugera millora al executar-lo amb diferents processadors. Aquesta millora és molt petita ja que com podem veure en el diagrama el programa es descomposa en 2 tasques, una de molt petita i una altra de molt gran. Per obtenir millors resultats hauríem de millorar la descomposició i el tamany de les tasques per fer-ho més eficient. El paral·lelisme es de 1.000114, el que ens indica que el programa amb infinits processadors s'executa un 0.0114 més ràpid.



Versió 1

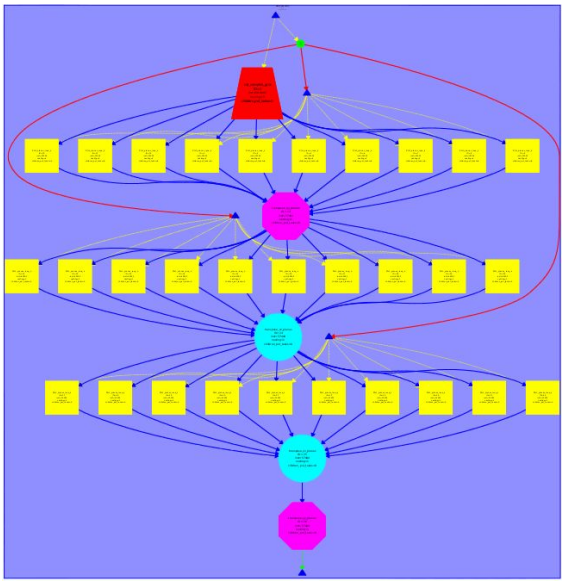
En aquesta versió, tot i que hem realitzat alguns canvis en el codi, podem veure que no es produeixen canvis en les xifres. Trobem la justificació en el diagrama de tasques. Com podem veure, s'han creat més tasques però es segueixen executant de forma seqüencial, el que provoca que no puguem aprofitar cap estrategia de paral·lelisme.

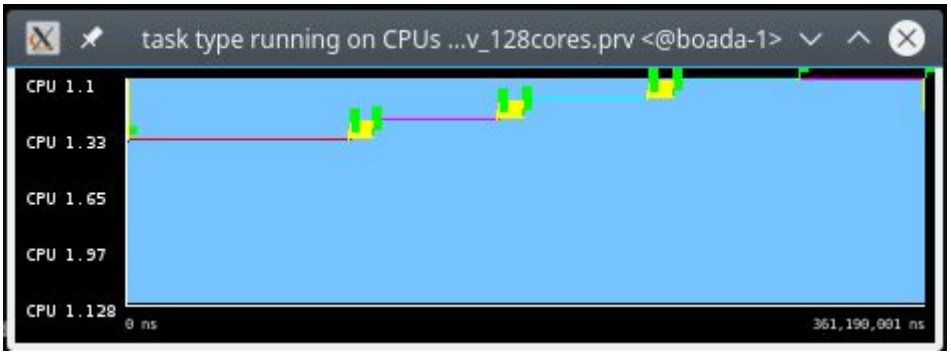
A partir d'ara, adjuntarem per a cada versió, una captura del diagrama de tasques corresponent i una captura del Paraver de l'execució del programa en infinits processadors, ja que per a 1 processador ja hem justificat que sempre és el mateix. En l'annex es troben captures del codi que hem usat per a executar cada versió.



Versió 2

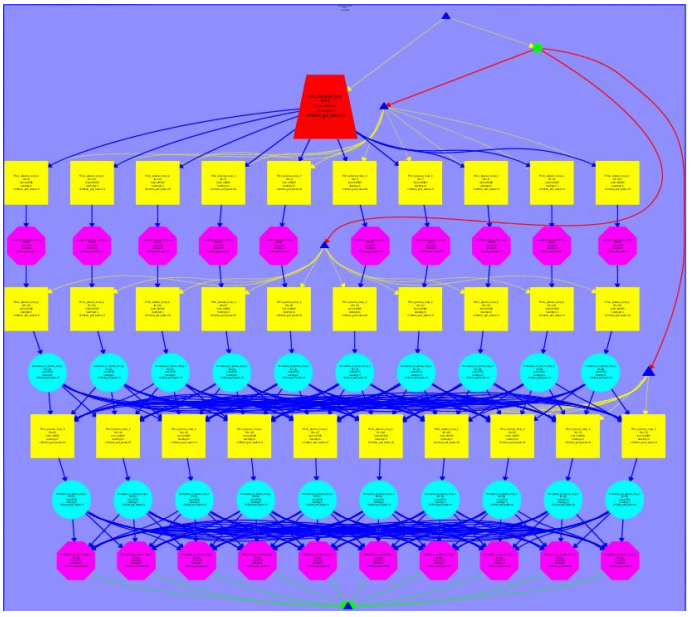
En aquesta versió veiem una primera millora en el temps d'execució ja que com veiem en el gràfic hem dividit la tasca ffts1_planes en tasques més petites i executades paral·lelament, per tant, aquest cop sí que hi ha una diferència. Hem implementat aquesta descomposició de tasques aplicant en el codi el que ens diu l'enunciat, és a dir, assignar una tasca a cada iteració del bucle k.

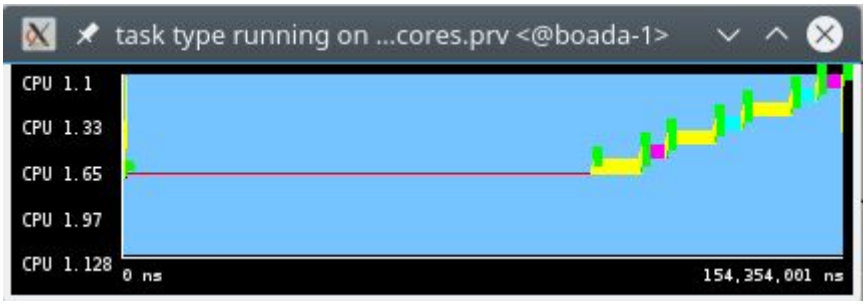




Versió 3

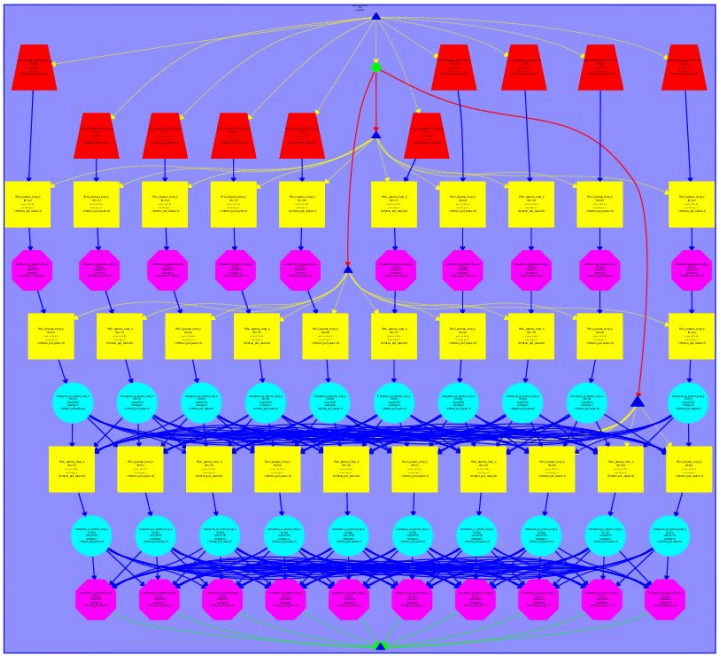
En aquesta versió, partim de la millora anterior, per tant, obtindrem una millora encara major. Això és perquè hem aplicat la millora anterior a les funcions transpose_xy_planes i transpose_zx_planes. Hem assignat a cada una de les iteracions del bucle k una tasca diferent executable paral·lelament.

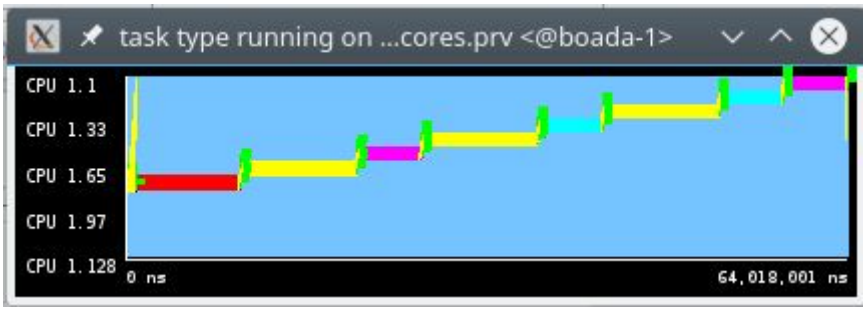




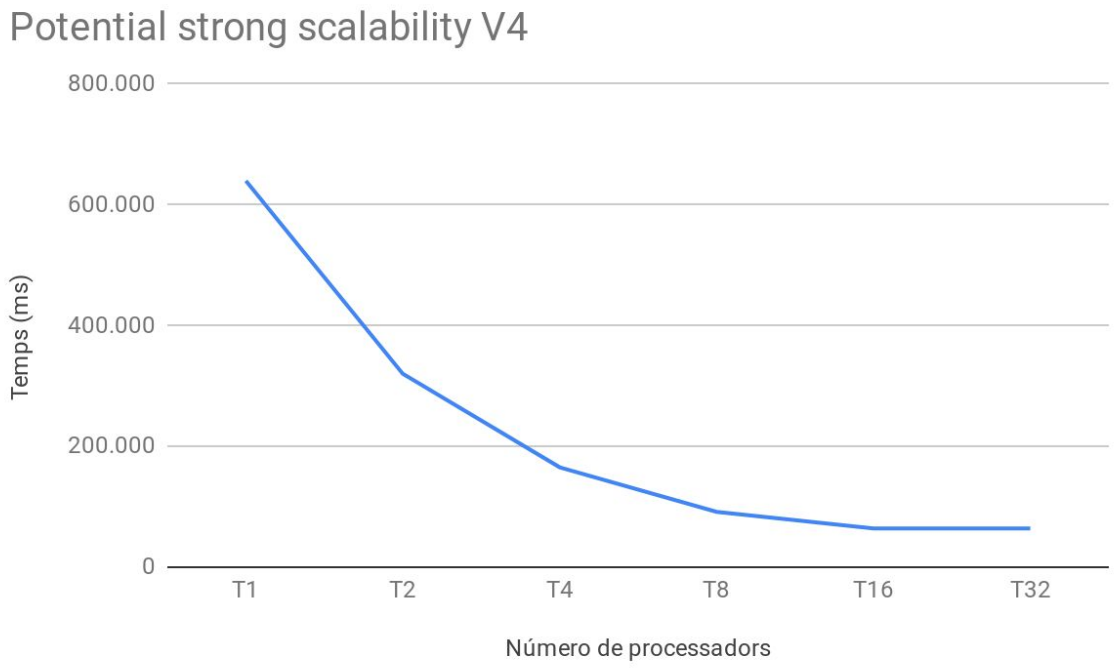
Versió 4

En aquesta versió aplicarem la millora que hem vist a les anteriors versions en la última funció que queda sense paral·lelitzar. Estem parlant de la funció init_complex_grid. Això resulta en una millora encara major del temps que havíem obtingut anteriorment.



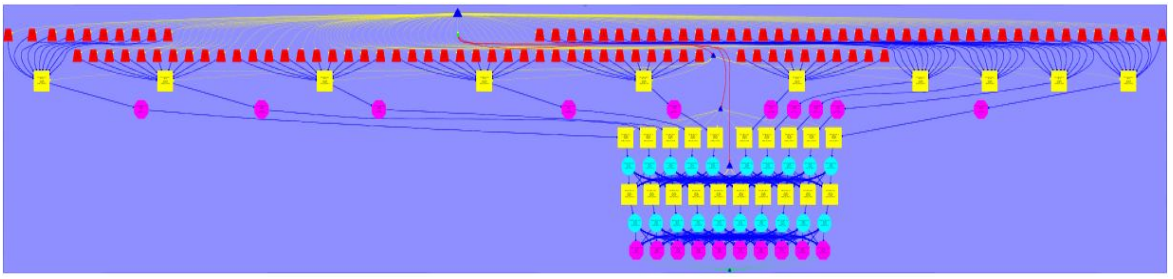


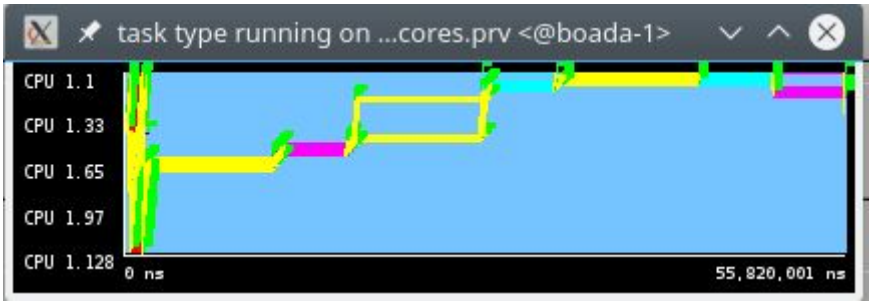
T1	639.780 ms
T2	320.310 ms
T4	165.389 ms
T8	91.496 ms
T16	64.018 ms
T32	64.018 ms



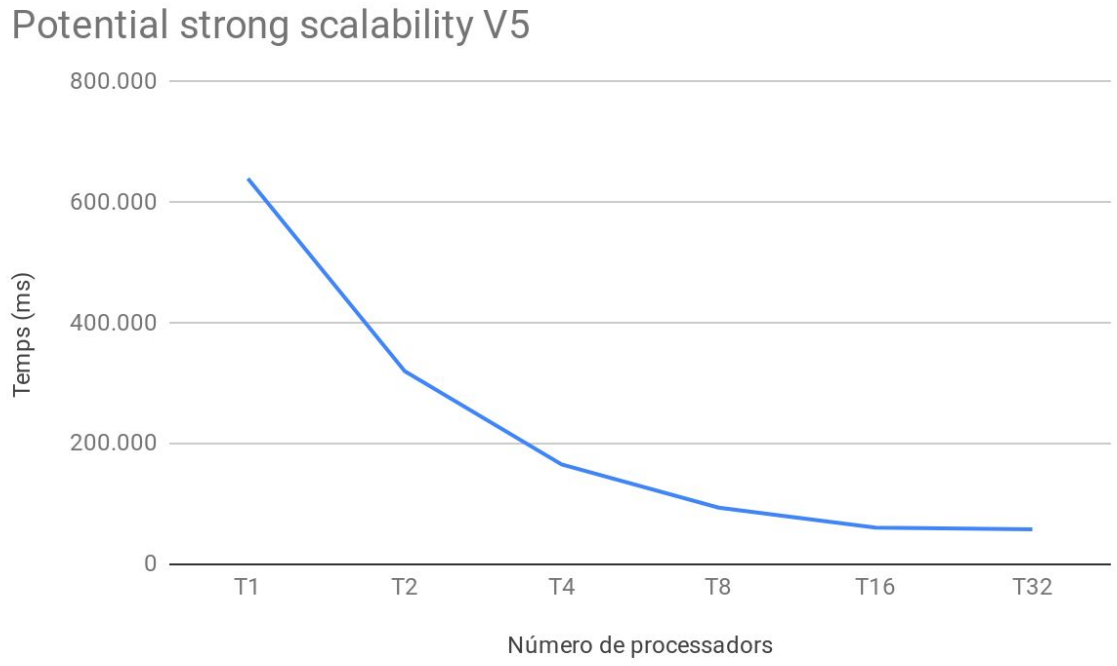
Versió 5

Finalment, acabarem la optimització aplicant una última descomposició de tasques a la funció init_complex_grid altre cop. Aquesta vegada, en comptes de paral·lelitzar el bucle k, paral·lelitzarem el bucle j, que executa més iteracions durant la total execució del programa. Això resultarà en una millora respecte la versió anterior però no tan gran com les que havíem anat veient.





T1	639.780 ms
T2	320.081 ms
T4	165.721 ms
T8	94.020 ms
T16	60.913 ms
T32	57.928 ms



Tot i que els plots de escalabilitat de les versions 4 i 5 són semblants, podem apreciar la millora de temps deguda a la major paral·lelització del codi. Aquesta major paral·lelització és apreciable si mirem els dos diagrames de tasques on veiem que V5 divideix la funció init_complex_grid en moltes més tasques executables a la vegada. Per aquesta raó, V5 és capaç d'aprofitar molt millor els recursos donats, el que fa que T_∞ sigui més gran que T32, quan a V4, T_∞ = T16.

Sessió 3

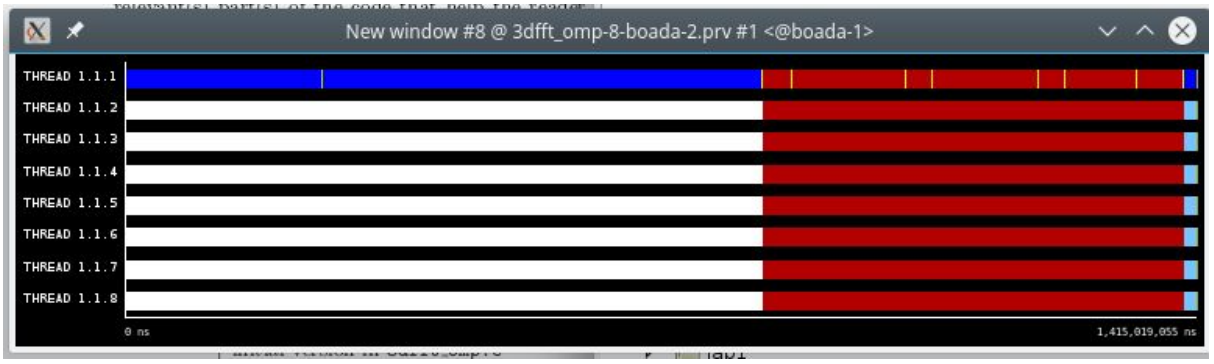
Understanding the parallel execution of 3DFFT

Version	φ	S_∞	T_1	T_8	S_8
initial version in 3dfft_omp.c	0.62	2.63	2328188 μs	1415019μs	1.64
new version with improved φ	0.91	11.1	2319896μs	986759μs	2.35
final version with reduced parallelisation overheads	0.92	13.01	2127374μs	452718μs	4,7

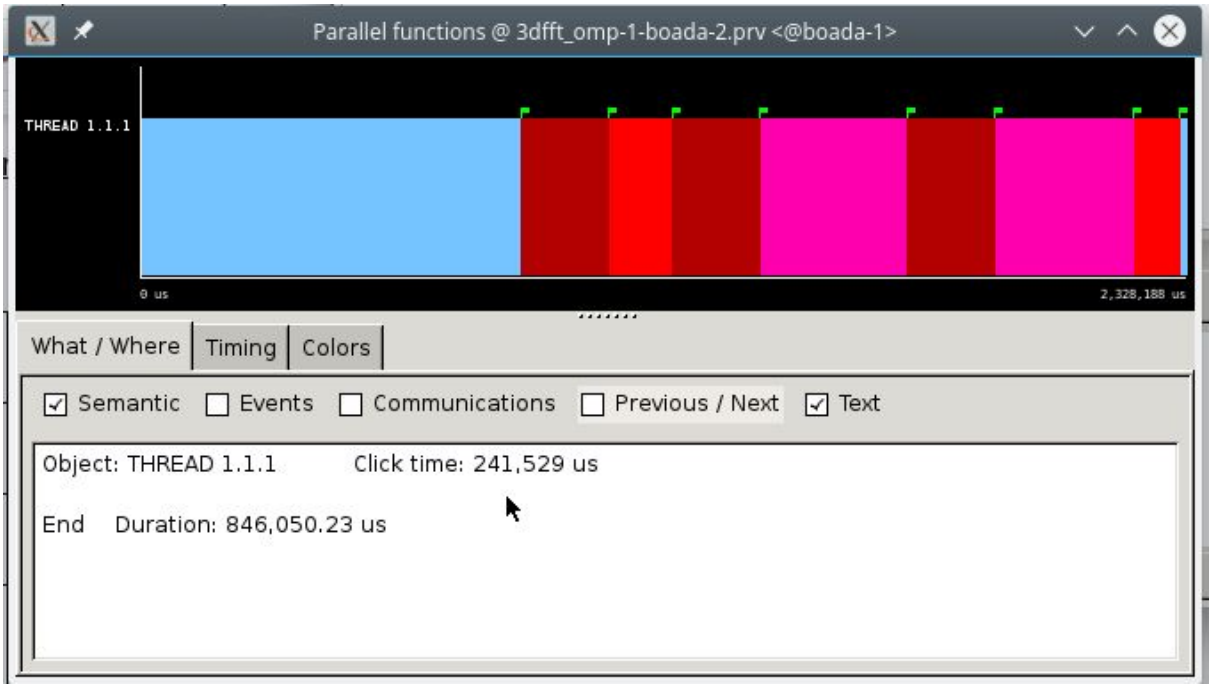
Tseq = 884711μs
Tpar = 1443476 μs
Tseq millorada I = 208057μs
Tpar millorada I = 2111839μs
Tseq millorada II = 201388μs // paraver -> una mica més gran que el real
Tpar millorada II = 2419595μs

Obtenim T1 a partir de la següent imatge, serà una mica mes gran que el real.

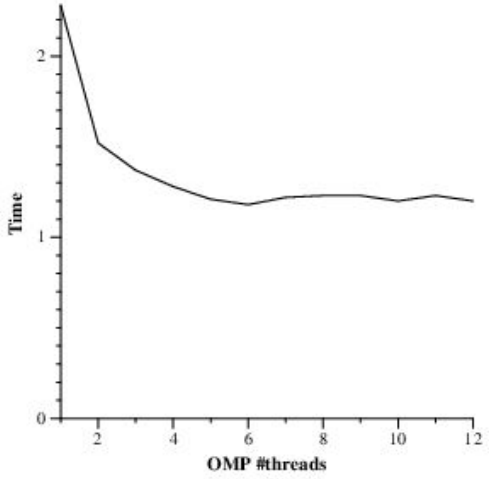
Obtenim T8 a partir de la següent imatge. Representa el temps d'execucio usant 8 processadors.

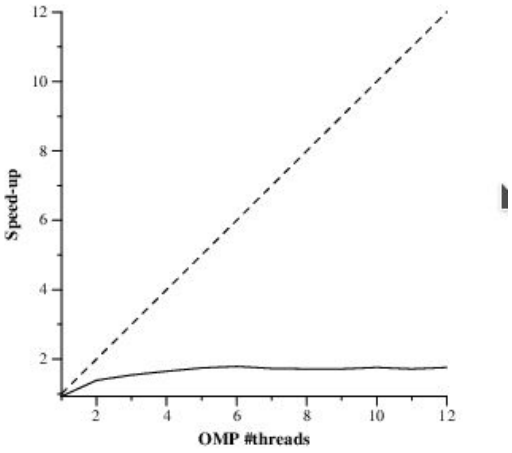


Obtenim Tseq sumant la durada de les dues zones blaves que es veuen a continuació. Aquestes zones representen la part no paral.lelitzable del codi.



Un cop tenim Tseq i Ttotal podem calcular φ = Tpar/ Ttotal sabent que Ttotal = Tseq + Tpar.

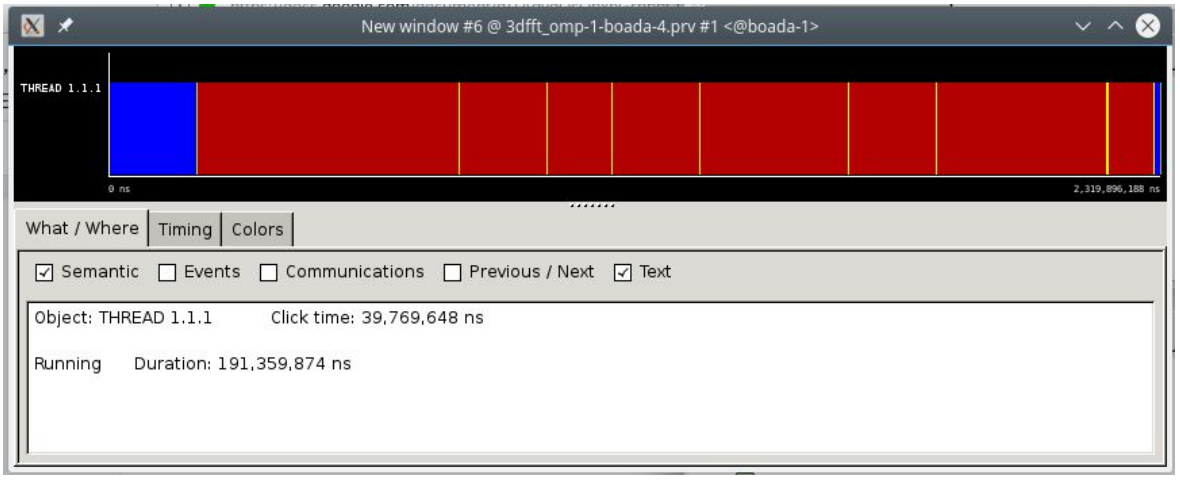




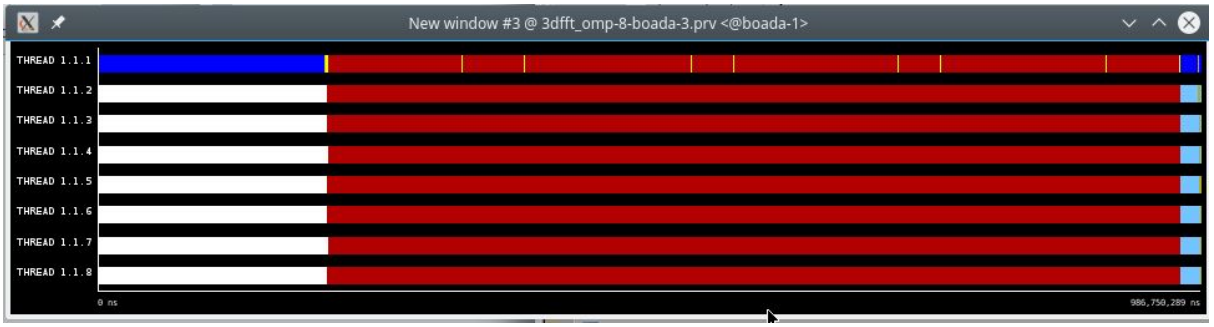
Com veiem, aquesta versió del programa és poc paral·lelitzable ja que està molt lluny de la situació ideal. Això es deu a que hi ha una gran part del programa que no es pot paral·lelitzar, cosa que ens afecta críticament al speed-up tal i com es veu en la gràfica.

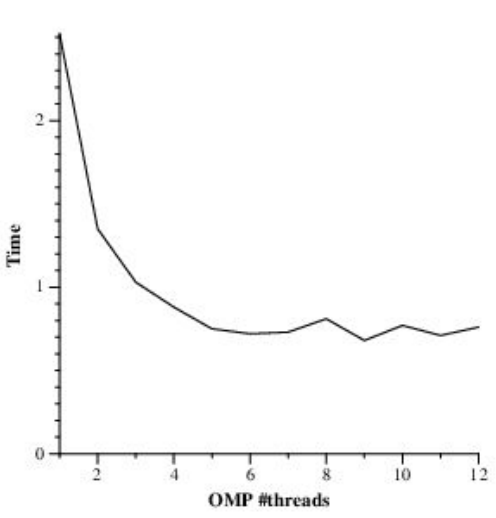
Versió Millorada I

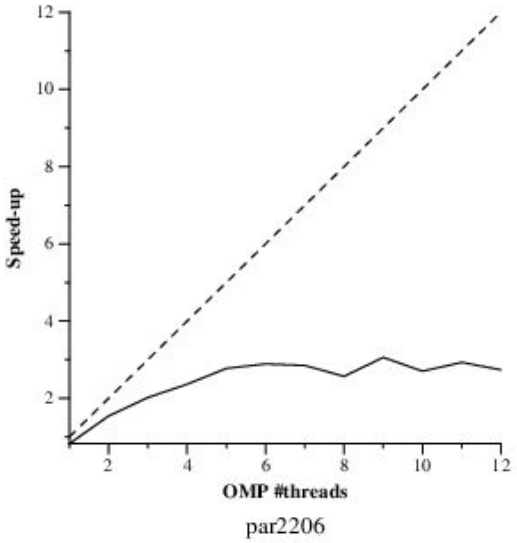
Obtenim T1 a partir de la informació de la següent captura. També podem obtenir Tseq sumant la duració de les zones blaves que és la duració de la part no paral·lelitzable del programa.



Obtenim T8 a partir de la següent captura.



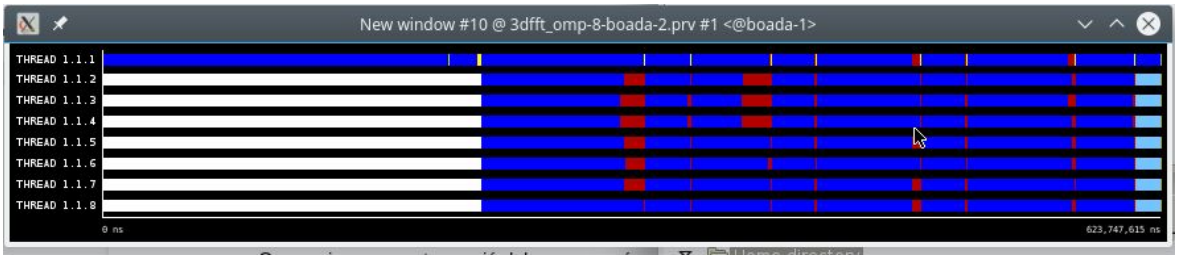




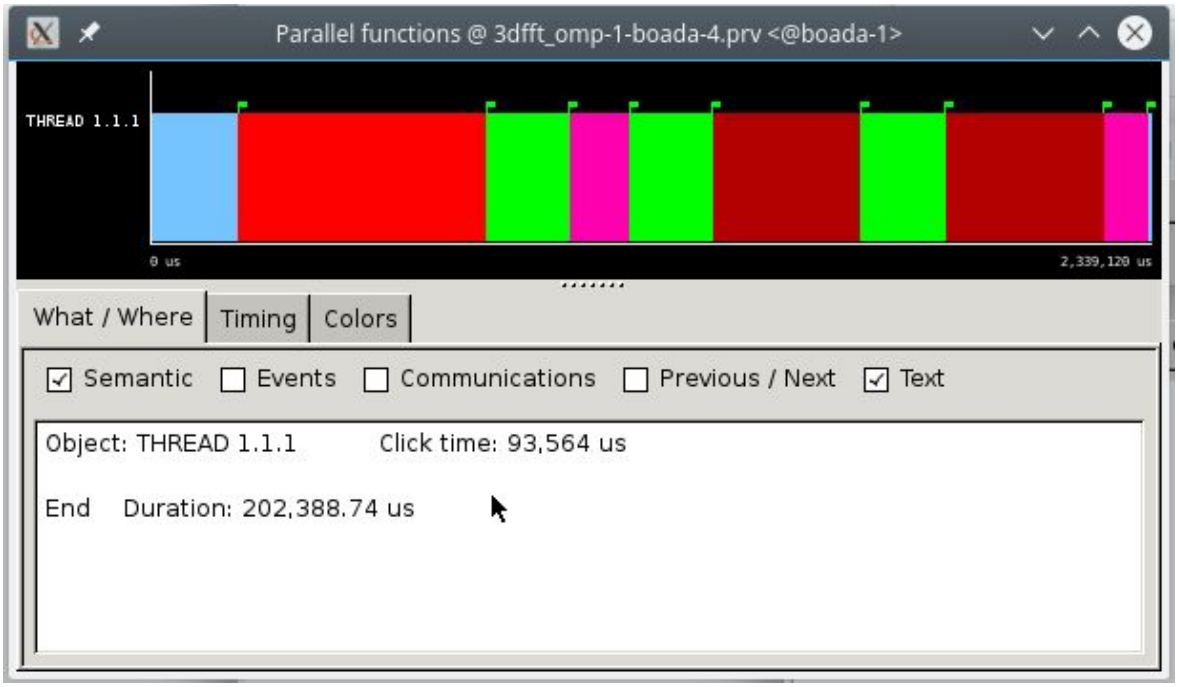
Al reduir el Tseq millora el speed-up

Versió Millorada II

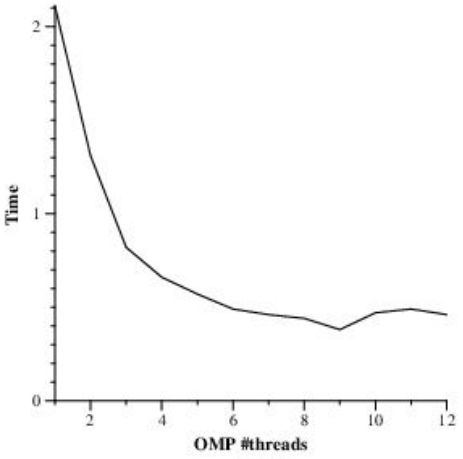
Hem obtingut T1 amb l'ajut de la comanda time des del terminal.
Obtenim T8 a partir de la següent captura



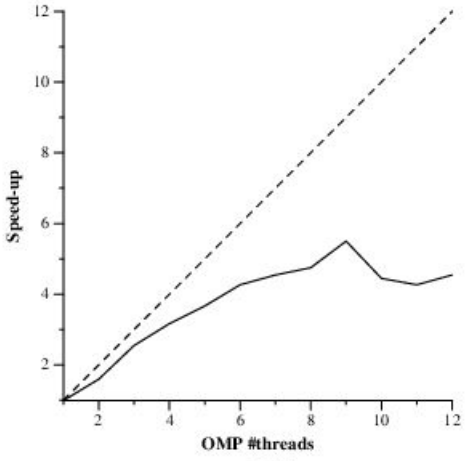
Obtenim Tseq i Tpar usant paraver.



Al fer el càlcul de φ hem usat els resultats obtinguts des de paraver ja que ens permetia obtenir Tseq i Tpar



par2206
Min elapsed execution time
Wed Feb 26 12:58:15 CET 2020



par2206
Speed-up wrt sequential time
Wed Feb 26 12:58:15 CET 2020

En aquest punt podem veure que el programa és més paral·lelitzable ja que s'acosta més a la situació ideal, això es deu a que hem creat tasques més petites que han afavorit a la paralelització del programa.

Annex

A continuació incloem captures del codi que hem fet servir a cada versió del programa en la Sessió 2.

```
void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N]) {
    int k,j;

    for (k=0; k<N; k++) {
     tareador_start_task("ffts1_planes_loop_k");
     for (j=0; j<N; j++) {
       fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0], (fftwf_complex *)in_fftw[k][j][0]);
     }
     tareador_end_task("ffts1_planes_loop_k");
    }
}
```

Versió 2

```
void transpose_xy_planes(fftwf_complex  tmp_fftw[][N][N], fftwf_complex in_fftw[][N][N]) {
    int k,j,i;

    for (k=0; k<N; k++) {
    tareador_start_task("transpose_xy_planes_loop_k");
     for (j=0; j<N; j++) {
       for (i=0; i<N; i++)
       {
         tmp_fftw[k][i][j][0] = in_fftw[k][j][i][0];
         tmp_fftw[k][i][j][1] = in_fftw[k][j][i][1];
       }
     }
    tareador_end_task("transpose_xy_planes_loop_k");
   }
}

void transpose_zx_planes(fftwf_complex in_fftw[][N][N], fftwf_complex tmp_fftw[][N][N]) {
    int k, j, i;

    for (k=0; k<N; k++) {
     tareador_start_task("transpose_zx_planes_loop_k");
     for (j=0; j<N; j++) {
       for (i=0; i<N; i++)
       {
         in_fftw[i][j][k][0] = tmp_fftw[k][j][i][0];
         in_fftw[i][j][k][1] = tmp_fftw[k][j][i][1];
       }
     }
     tareador_end_task("transpose_zx_planes_loop_k");
    }
}
```

Versió 3

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

  for (k = 0; k < N; k++) {
    tareador_start_task("init_complex_grid_loop_k");
    for (j = 0; j < N; j++) {
      for (i = 0; i < N; i++)
      {
        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }

    }
   tareador_end_task("init_complex_grid_loop_k");
  }
}
```

Versió 4

```
void init_complex_grid(fftwf_complex in_fftw[][N][N]) {
  int k,j,i;

  for (k = 0; k < N; k++) {
    for (j = 0; j < N; j++) {
      tareador_start_task("init_complex_grid_loop_j");
      for (i = 0; i < N; i++)
      {
        in_fftw[k][j][i][0] = (float) (sin(M_PI*((float)i)/64.0)+sin(M_PI*((float)i)/32.0)+sin(M_PI*((float)i/16.0)));
        in_fftw[k][j][i][1] = 0;
#if TEST
        out_fftw[k][j][i][0]= in_fftw[k][j][i][0];
        out_fftw[k][j][i][1]= in_fftw[k][j][i][1];
#endif
      }
      tareador_end_task("init_complex_grid_loop_j");
    }
  }
}
```

Versió 5