PARAL·LELISME: LABORATORI 1

Pol Casacuberta Gil i Salvador Grimalt BInimelis
Quadrimestre 1, curs 22-23
Paral·lelisme, grup 21

Lab1: Experimental setup and tools

Sessió 1: Experimental setup

A la primera sessió se centra en conèixer les eines que farem servir aquest quadrimestre a les classes de laboratori; entre elles tenim a boada, un servidor del departament d'Arquitectura de Computadors, format per diversos nodes, tenint de boada-6 a boada-8 que són interactius (els programes s'hi executaran instantàniament) i de boada-11 a boada-14 que fan servir el sistema de cues (s'executen els programes d'un en un).

Node architecture and memory

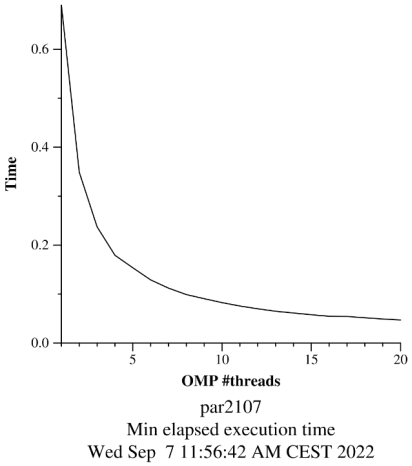
Primer de tot, va tocar veure com és l'arquitectura dels nodes de boada utilitzant els comandos lscpu i lstopo, que generen tres fitxers .fig:

	Any of the nodes among boada-11 to boada-14
Number of sockets per node	2
Number of cores per socket	10
Number of threads per core in node	2
Maximum core frequency	3200
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	1 MB
Last-level cache size (per-socket)	3
Main memory size (per socket)	2,35 GB
Main memory size (per node)	23,5 GB

Strong vs. weak scalability for Pi

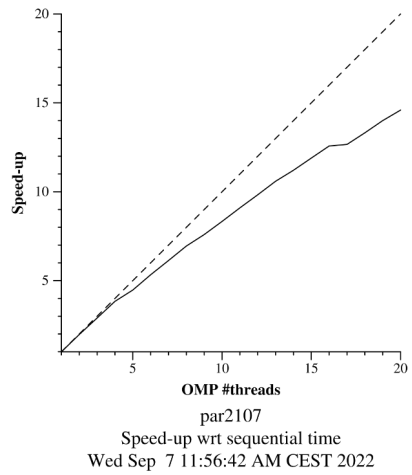
Com hem dit abans, tenim dos tipus d'execució, *interactive* i *queued*, on triarem entre executar paral·lelament amb altres programes (run-xxxx.sh) o utilitzant el sistema de cues (sbatch [-p partition] ./submit-xxxx.sh), respectivament. A continuació tenim els resultats d'executar pi_seq.c

#threads	Interactive: Timing information				Queued: Timing information			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1	2.36	0.00	0:02.37	99%	0.68	0.00	0:00.70	97%
4	2.36	0.01	0:01.19	199%	0.69	0.01	0:00.19	367%
8	2.39	0.05	0:01.23	199%	0.74	0.01	0:00.11	656%
16	2.41	0.13	0:01.28	199%	0.79	0.00	0:00.07	1138%
20	2.46	0.12	0:01.29	199%	0.83	0.00	0:00.06	1353%



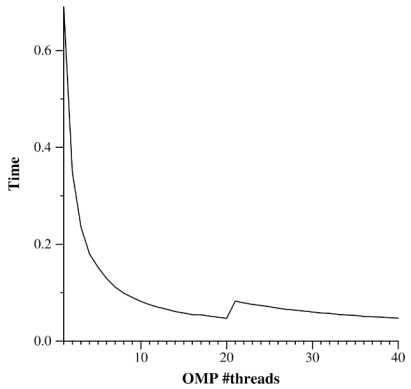
Strong Scalability

Diem que l'escalabilitat forta és la capacitat del sistema per a augmentar el nombre de threads mantenint la mida del problema reduint la càrrega de treball de cada processador. Sabem que els programes no són mai paral·lelitzables a l'infinit pels overheads per la creació de tasques i sincronització. Això vol dir que si incrementem massa el nombre de threads, arribarà a un punt que afegir nous processadors no afectarà al temps d'execució, ja que hem arribat al nivell màxim de paral·lelització.

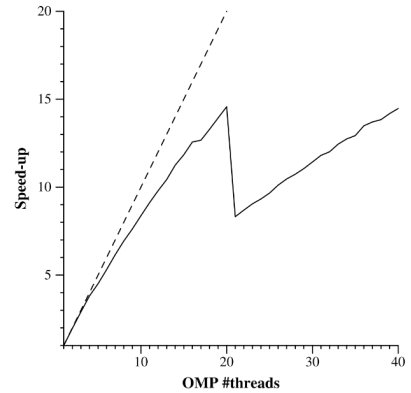


A l'esquerra tenim els resultats d'executar pi_seq a boada-7 utilitzant el mètode de strong scability on es demostra el que hem explicat abans.

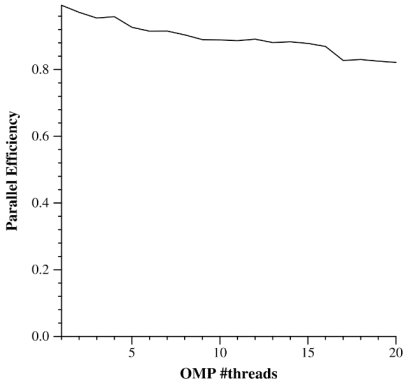
La mida del problema és de 1.000.000.000 iteracions, i veiem que, si incrementem el nombre de threads, el temps es redueix logarítmicament i si mirem les darreres parts (de 15 a 20 threads) i veiem que la línia del temps d'execució es va tornant horitzontal.



par2107
Min elapsed execution time
Wed Sep 7 12:03:16 PM CEST 2022



par2107
Speed-up wrt sequential time
Wed Sep 7 12:03:16 PM CEST 2022



par2107
Parallel Efficiency w.r.t. one thread (weak scaling)
Wed Sep 7 12:26:34 PM CEST 2022

Ara en comptes de mirar fins a 20 threads, a les figures de l'esquerra, mirarem fins a 40 i veiem un comportament estrany on veiem que va decreixent de l'1 al 20, però del 20 al 21 de sobte creix. Això és degut al hyper-threading, que fa que el cost de fer-lo és major que el benefici que obtenim.

Weak Scalability

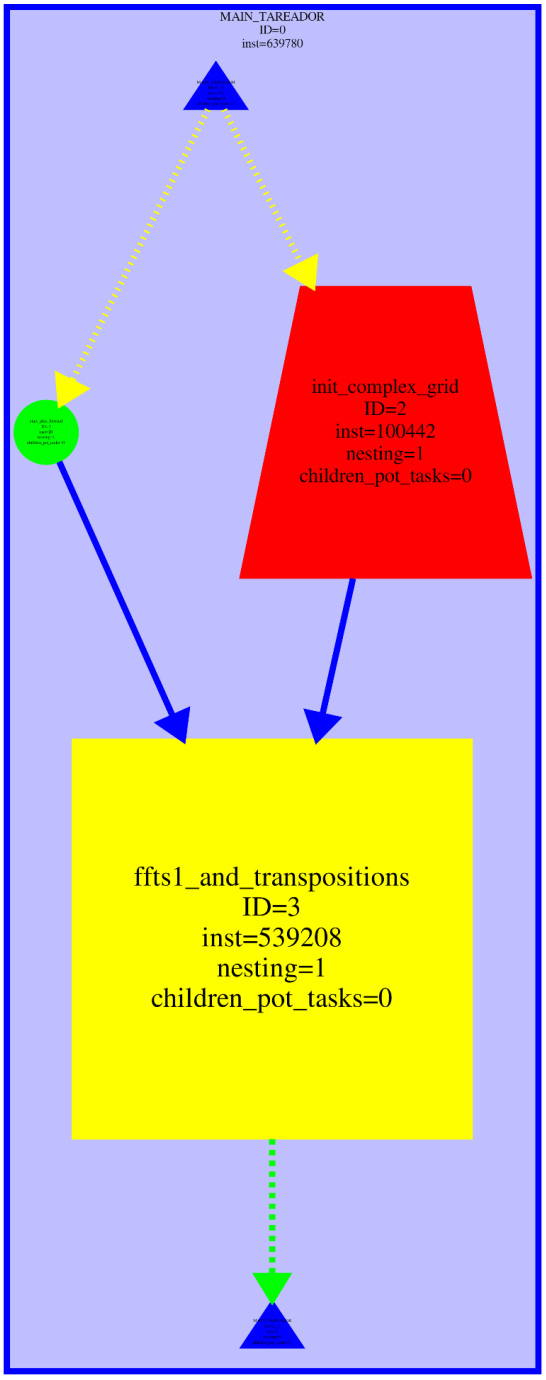
A Weak Scalability fem el contrari. Ara aprofitarem els recursos computacionals que obtenim paral·lelitzant per a incrementar la mida del problema proporcionalment al nombre de threads. A la figura de sota a l'esquerra veiem que l'speed up es manté més o menys constant mentre la càrrega de feina puja.

A la figura de sota a l'esquerra el que observem és com l'speed up varia relativament poc (de 1.0 a 0.8 d'eficiència de la paral·lelització) multiplicant per 20 el nombre de threads, és a dir, augmentant molt la feina feta en aquest mateix interval de temps.

Analitzar sistemàticament descomposicions amb el Tareador

Hem estat investigant el que el Tareador ens pot oferir. Aquest és el que aprofitarem per trobar el paral·lelisme que podem aconseguir entre tasques.

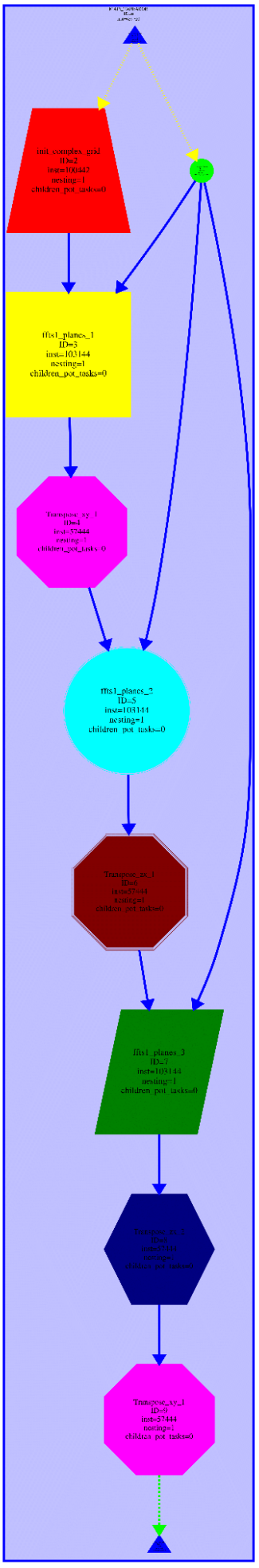
Per veure les diferents versions paral·lelitzades del programa 3dfft_tar.c hem modificat aquest programa, la primera versió del qual tenim a la dreta. Veiem que dues tasques són molt grans i la resta són molt més petites, això ens indica que són les grans les que volem paral·lelitzar.



Versió 1

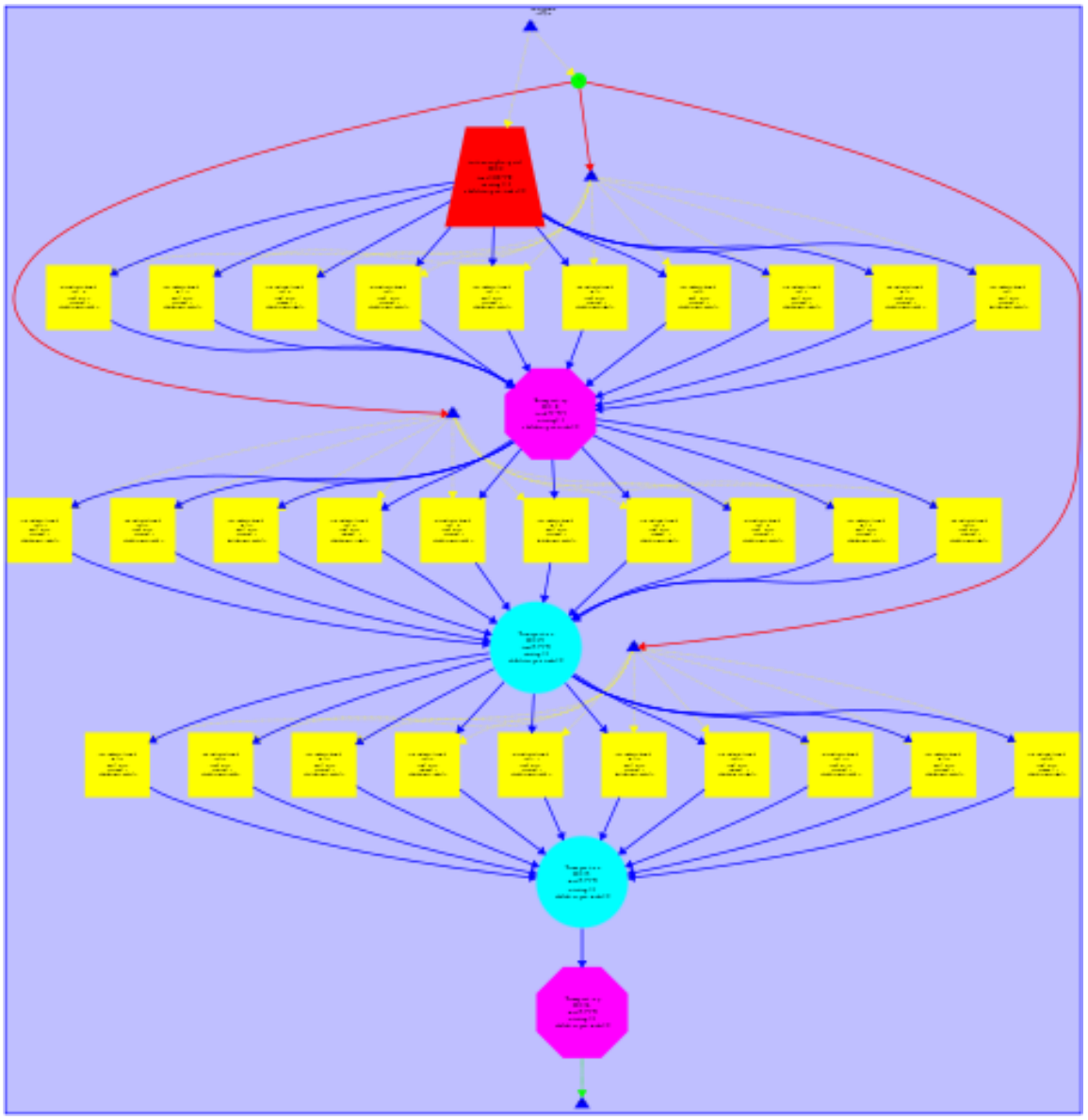
A la primera versió hem decidit dividir la tasca gran (ffts1_and transpositions) en unes quantes de més petites i granulades, una per cada funció d'aquesta.

Un cop modificat el codi veiem que el graf de dependències queda com el de la dreta, el quadrat groc original ha quedat dividit entre més tasques més petites. L'execució segueix essent seqüencial, ja que no hi hem aplicat el paral·lelisme encara.



Versió 2

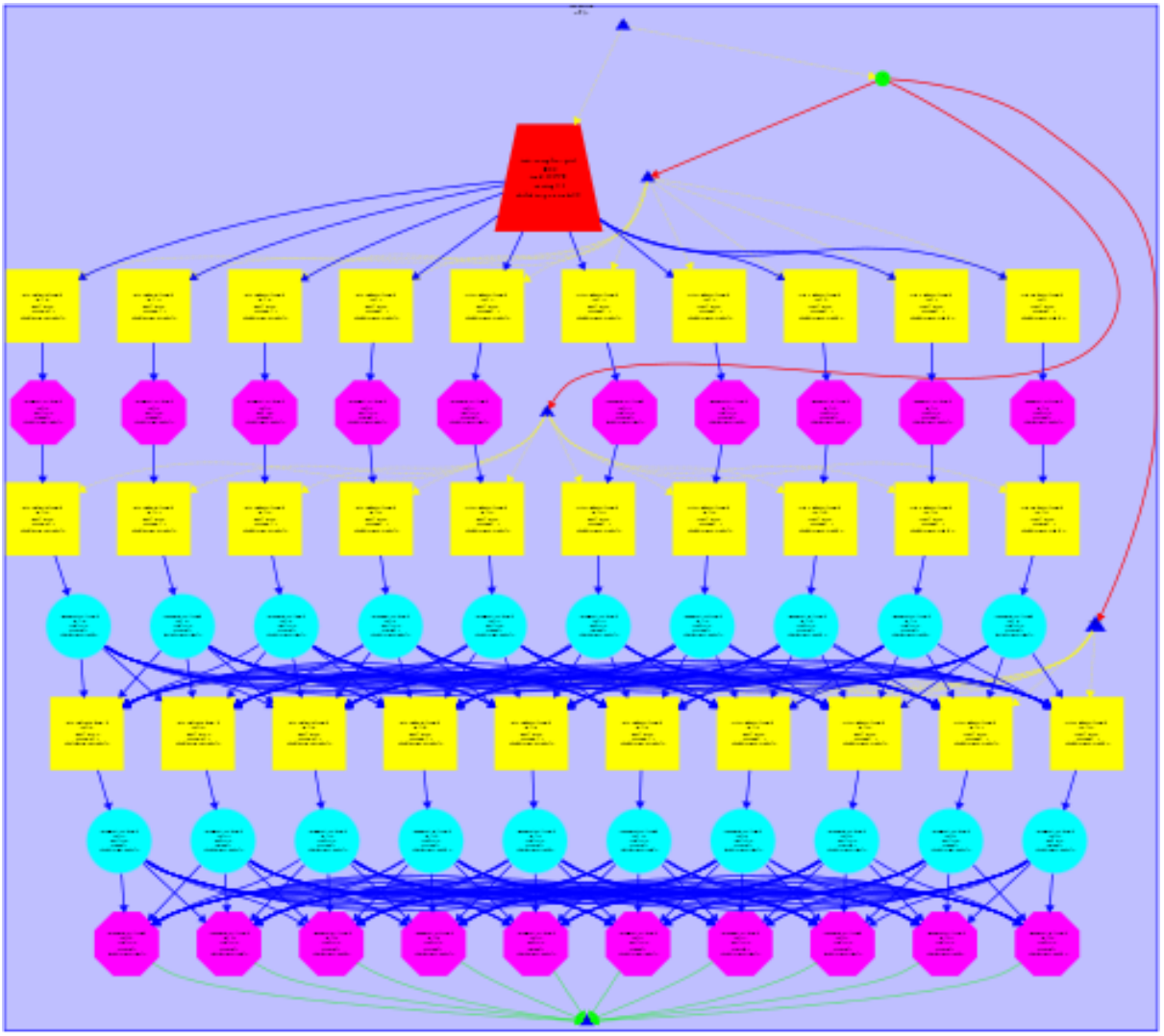
Ara, mantenint els canvis fets a la versió anterior, canviem la definició de les tasques invocant la funció ffts1_planes per paral·lelitzar les operacions de les files dels quadrats grocs. Aquest seria el graf de dependències que obtenim a través del tareador:



Amb aquests canvis, aconseguim paral·lelitzar en deu les tasques del quadrat groc (és a dir, les tasques del quadrat groc ara tarden un 10% del que tardaven abans).

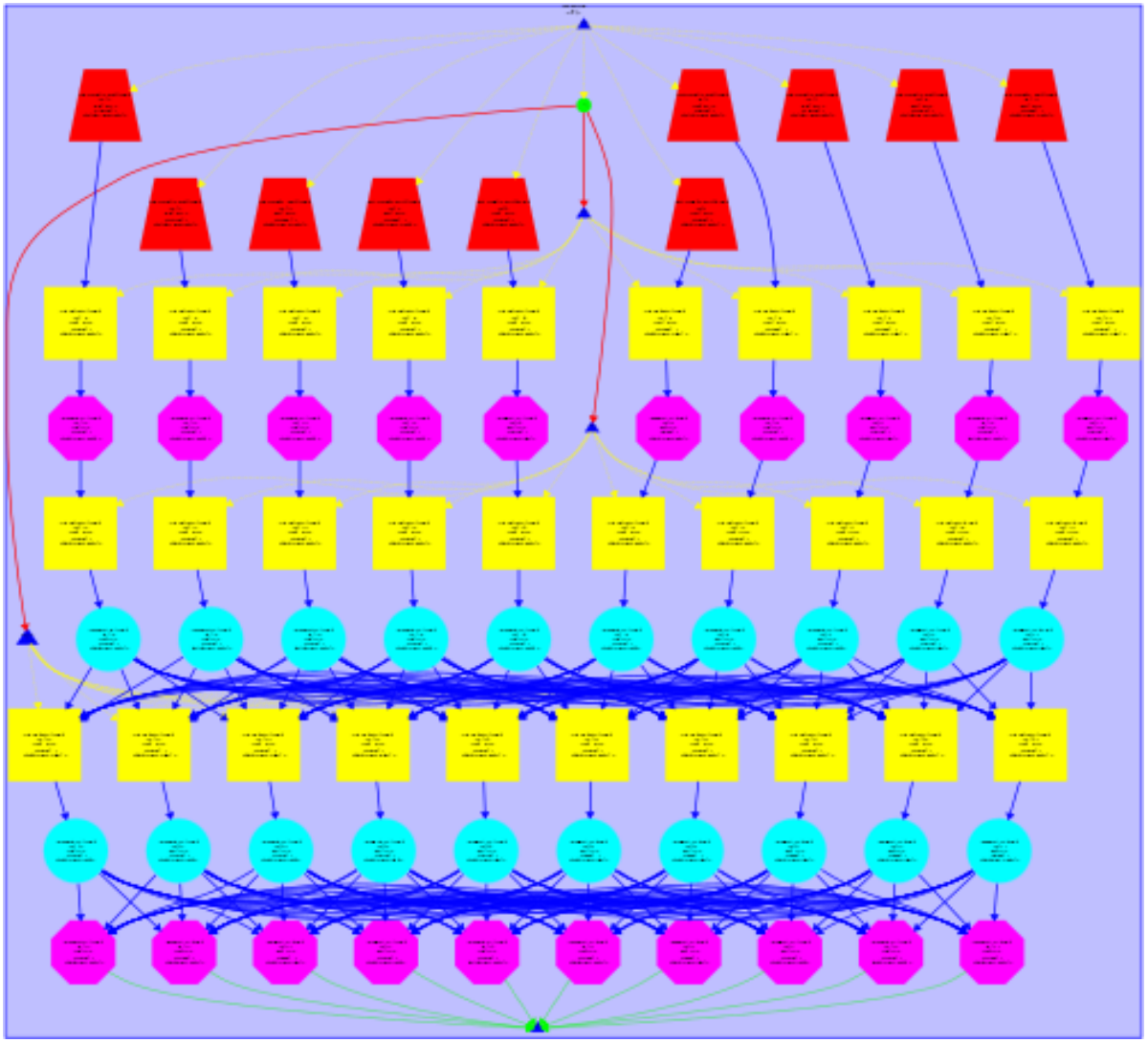
Versió 3

La tercera versió manté els canvis de la segona i consisteix en invocar les funcions de transpose_xy_planes i transpose_zx_planes al loop de la k, de manera similar a la versió anterior. El graf de dependències que aconseguim amb el tareador és aquest:



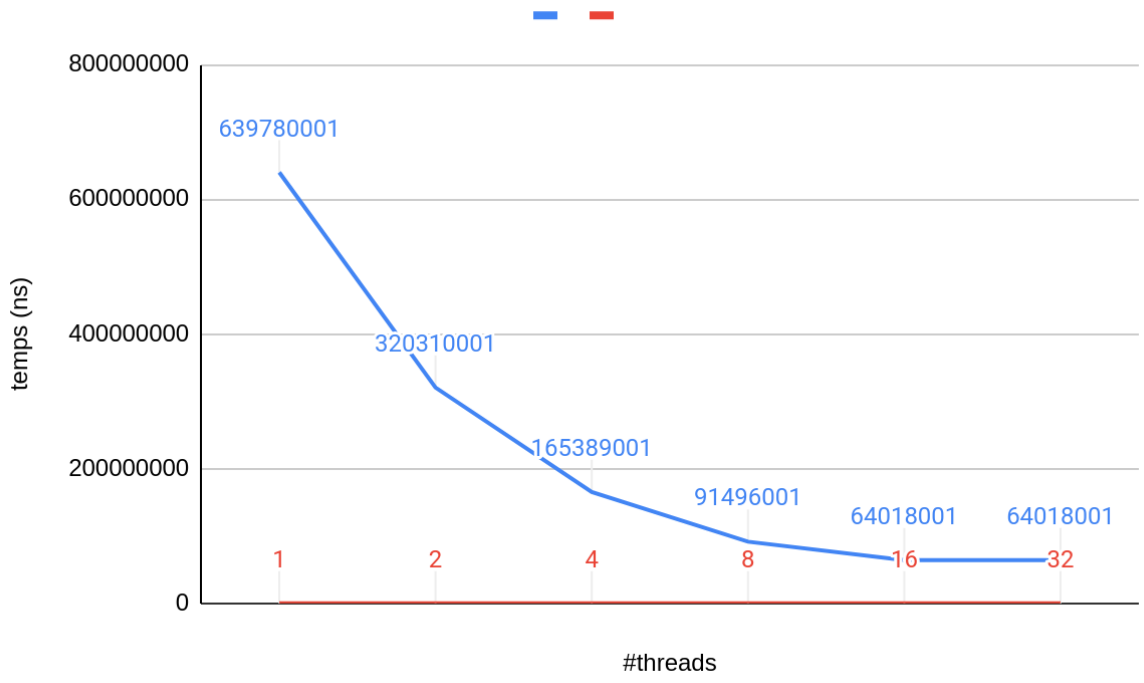
Versió 4

Aquesta versió consisteix a canviar el lloc de la definició de les tasques associades amb invocacions de init_complex_grid en tasques més petites dins de les funcions associades al k loop, com a la versió anterior.



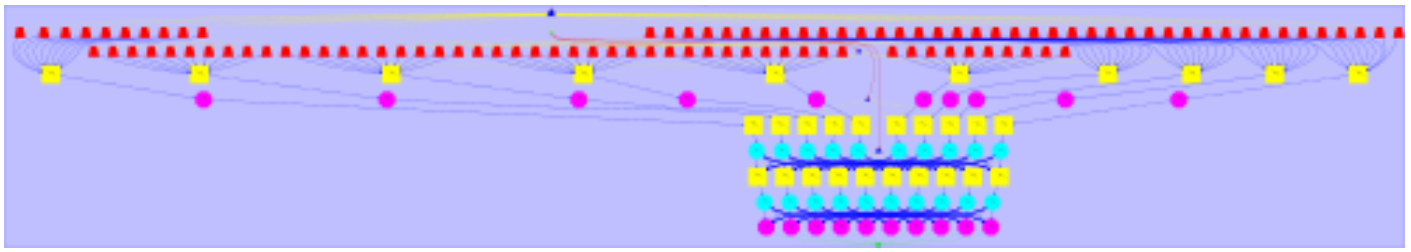
Si executem el programa per 2, 4, 8, 16 i 32 threads veiem que a mesura que afegim threads, anem escurçant el temps d'execució, tot i que l'escurçament no és linear respecte al nombre de threads, sinó que s'assembla més a una funció logarítmica que tendeix a un cert nombre (en aquest cas 64 018 001ns), això és per l'overhead que tenim degut a la sincronització entre threads.

#threads	temps
1	639 780 001 ns
2	320 310 001 ns
4	165 389 001 ns
8	91 496 001 ns
16	64 018 001 ns
32	64 018 001 ns

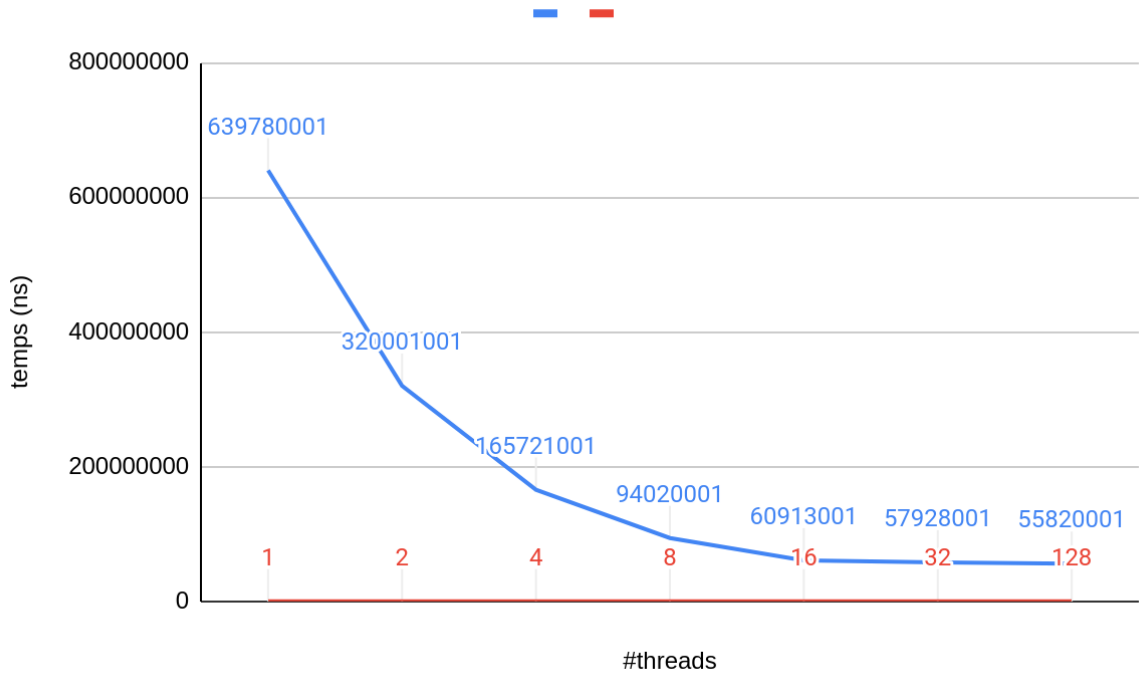


Versió 5

Aquesta és la versió final on ens quedarem amb tasques encara més concretes. El que hem fet és endinsar les tasques al loop j (dedins del loop k) la funció corresponent. El graf de dependències quedaria així:

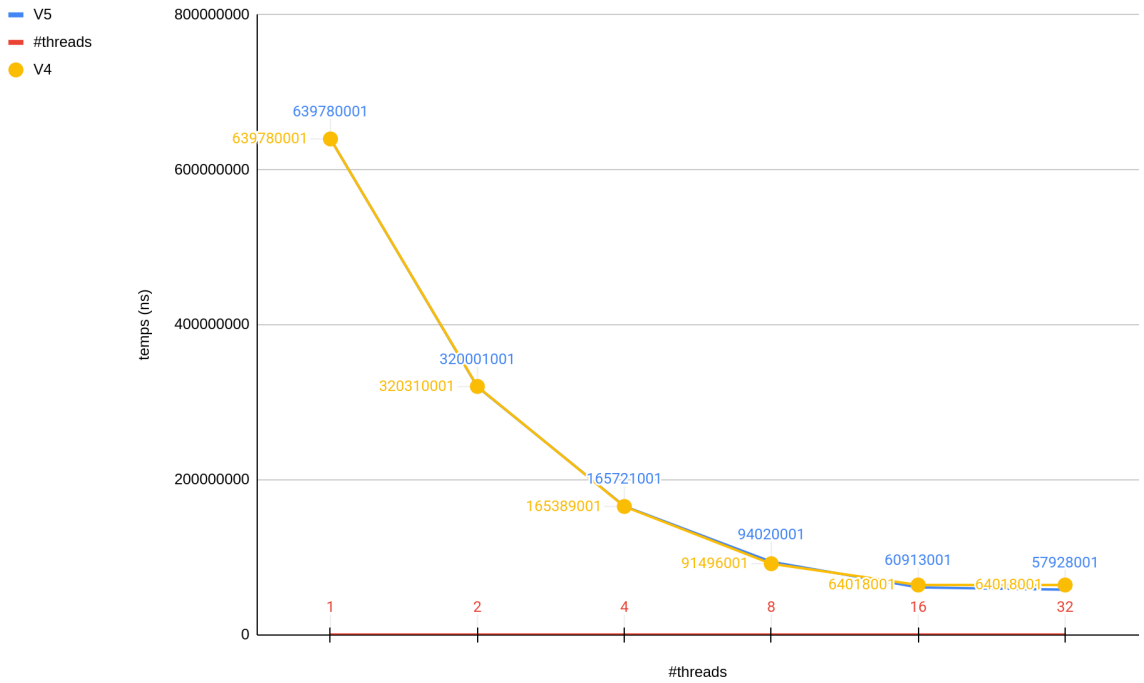


#threads	temps (ns)
1	639780001
2	320001001
4	165721001
8	94020001
16	60913001
32	57928001
128	55820001



V4 vs. V5

Ara comparem les versions 4 i 5 i, podem veure que fins als 16 threads la diferència és mínima però que a partir de llavors la versió 4 es queda estancada i la versió 5 segueix reduint el seu temps d'execució, això és perquè la versió 5 està millor granulada i això fa que un nombre major de threads doni millors resultats.

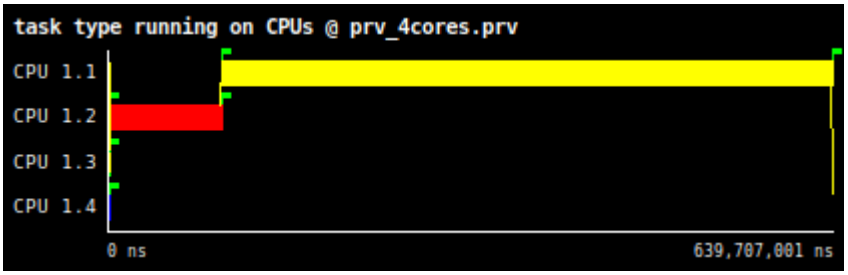


Conclusions

	T1	Tinf	Paral·lelisme
seq	639 780 001 ns	639 780 001 ns	1
v1	639 780 001 ns	639 780 001 ns	1
v2	639 780 001 ns	320 001 001 ns	1,999306249
v3	639 780 001 ns	165 389 001 ns	3,868334636
v4	639 780 001 ns	64 018 001 ns	10,50317651
v5	639 780 001 ns	57 928 001 ns	11,04439977

En aquesta veiem els diferents temps d'execució mínims de les nostres versions i podem dir que, excepte a la primera versió, totes incrementen el paral·lelisme.

Compilació i execució de programes OpenMP



Obtenint mètriques de paral·lelització fent ús de *modelfactors*

Ens pregunten les següents qüestions sobre l'output de modelfactors en l'execució del binari *3dfft_omp* mitjançant l'script *submit-strong-extrae.sh.*

L'escalabilitat és apropiada?
L'escalabilitat sembla prou pobra, ja que a partir de 16 processadors l'overhead és tan gran que triga més que amb un sol processador.

És l'overhead degut a la sincronització negligible?
L'overhead a mesura que augmenten els processadors, fent servir 16 processadors, arriba a ocupar fins a un 75% per a les tasques explícites per culpa de la sincronització. Sembla que no sigui pas negligible.

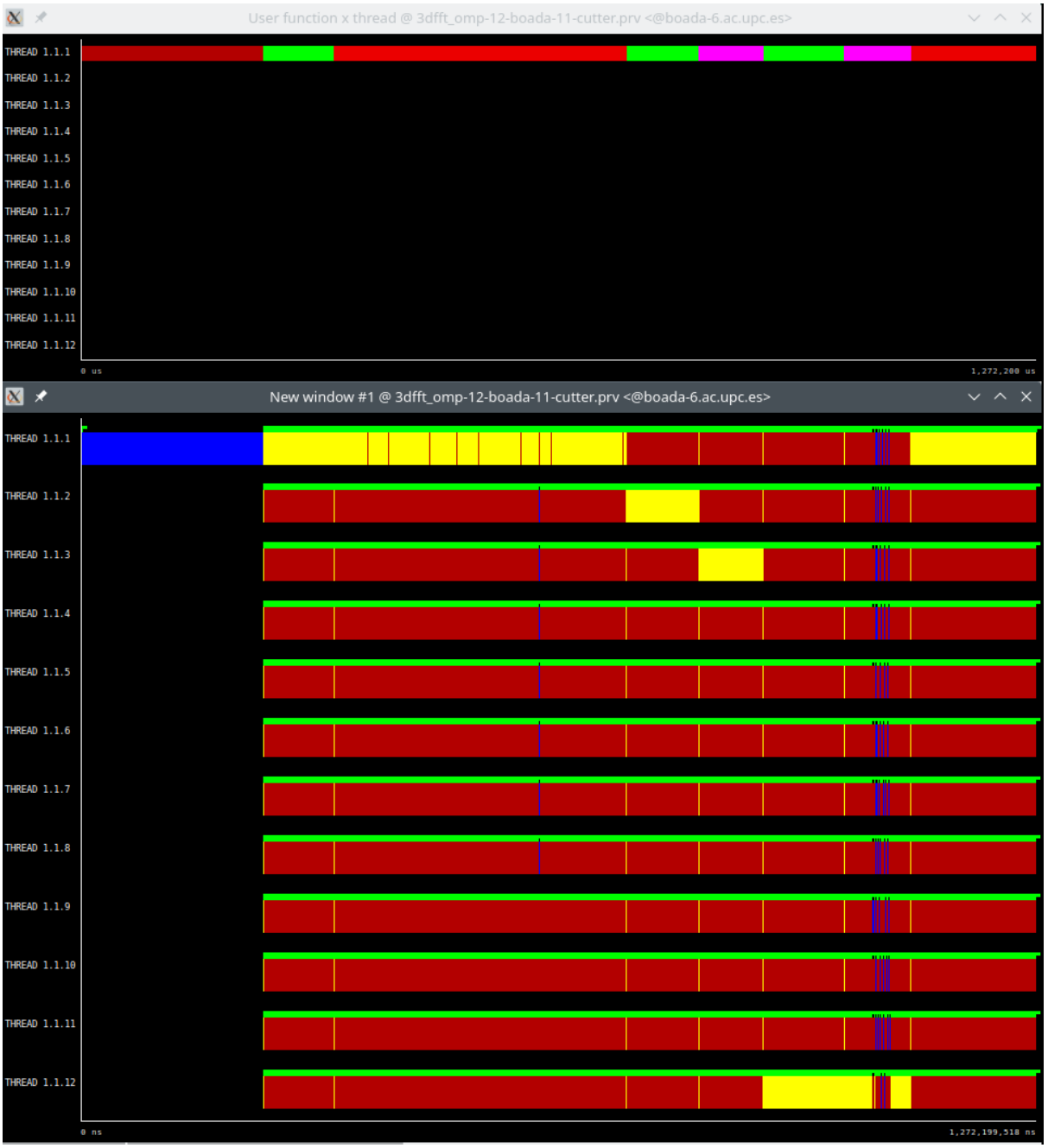
L'overhead està afectant el temps d'execució per les tasques explícites?
Està afectant en gran mesura al temps d'execució, ja que a mesura que augmentem els processadors veiem que inclús arriba a superar el temps amb un sol processador.

Quina és la fracció paral·lela per a aquesta versió del programa?
La fracció paral·lela correspon a un 82,92% del programa

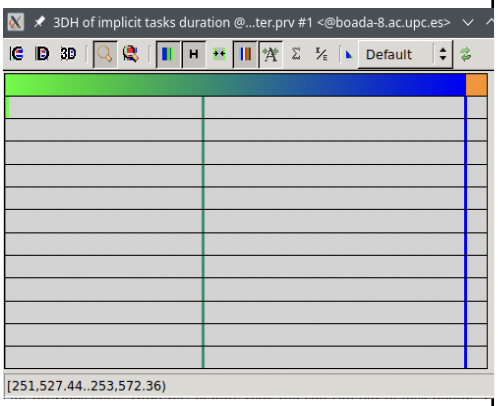
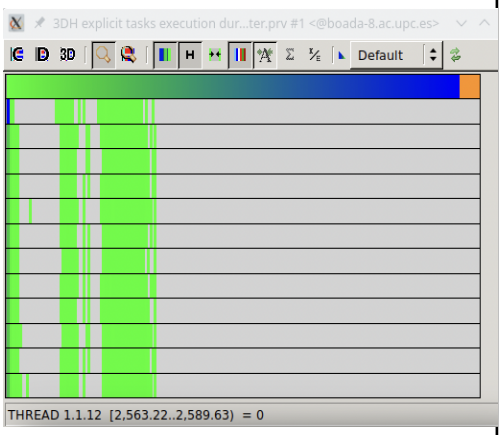
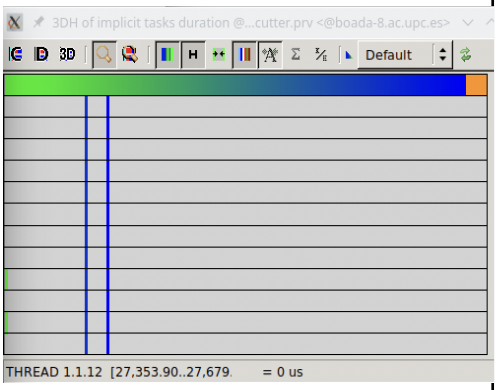
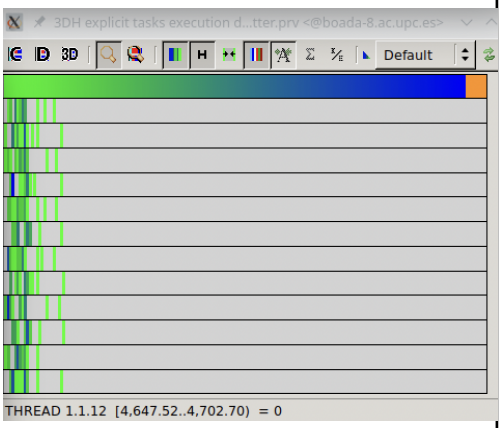
És l'eficiència per a les regions paral·leles adequada?
L'eficiència per a les fraccions paral·leles és sorprenentment baixa amb tan sols un 5,43% amb 16 processadors.

Quin és el factor que afecta més negativament?
Sembla evident que el que afecta més negativament no és pas que només es paral·lelitzi un 82,92% del programa sinó que hi ha molt overhead en aquesta paral·lelització.

Com s'ha dit en el document fet pel professorat de PAR en aquestes dues gràfiques sembla que hi hagi dos problemes prou evidents. La primera funció del programa no és paral·lela. I, en segon lloc, i segurament més important és que la major part dels threads estan majoritàriament fent tasques de sincronització això ho podem veure, ja que la major part de les tasques estan pintades de color vermell en la finestra de sota.
Semblaria ser que aquest overhead a més a més no és pas constant, pel fet que augmentant els processadors augmenta considerablement l'overhead per cada tasca explícita.



Tasques explícites i implícites

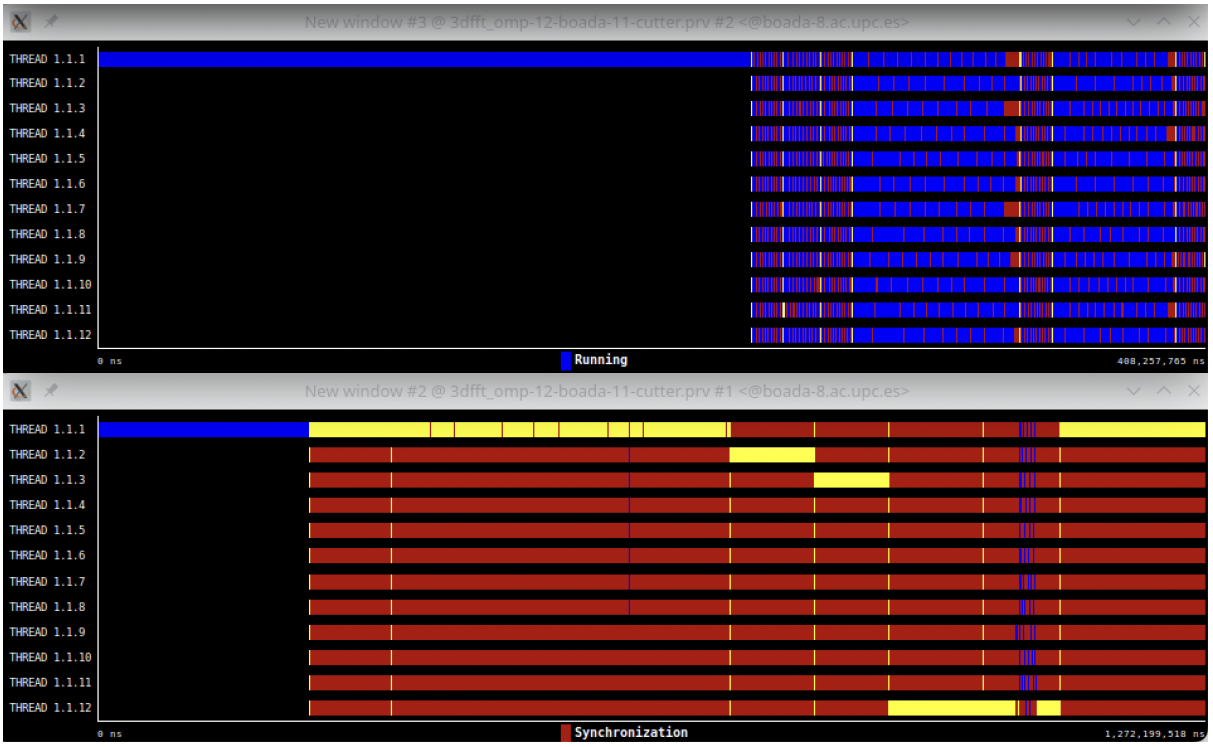
	Tasques implícites	Tasques explícites
Granularitat gruixuda		
Granularitat fina		

Per a veure quin tipus de granularitat tenim en les tasques explícites hem extret els histogrames del programa original sobre les tasques explícites i implícites, però per a poder-ho comparar amb quelcom que ens serveixi de referència hem fet servir també el programa amb l'optimització de l'overhead, fent servir un increment de la granularitat.
Per tant, en primera instància podem veure que tenim una granularitat prou gruixuda (ambdós figures de la part superior) en comparació amb la segona versió (figures de la part inferior).

Reduint overheads per paral·lelització i anàlisis

Se'ns proposa una millora que consisteix a incrementar la granularitat de les tasques comentant el taskloop interior de cada regió i descomentant el de fora. El canvi sembla haver millorat considerablement el temps d'execució passant de 1.272.199.518 ns a tan sols

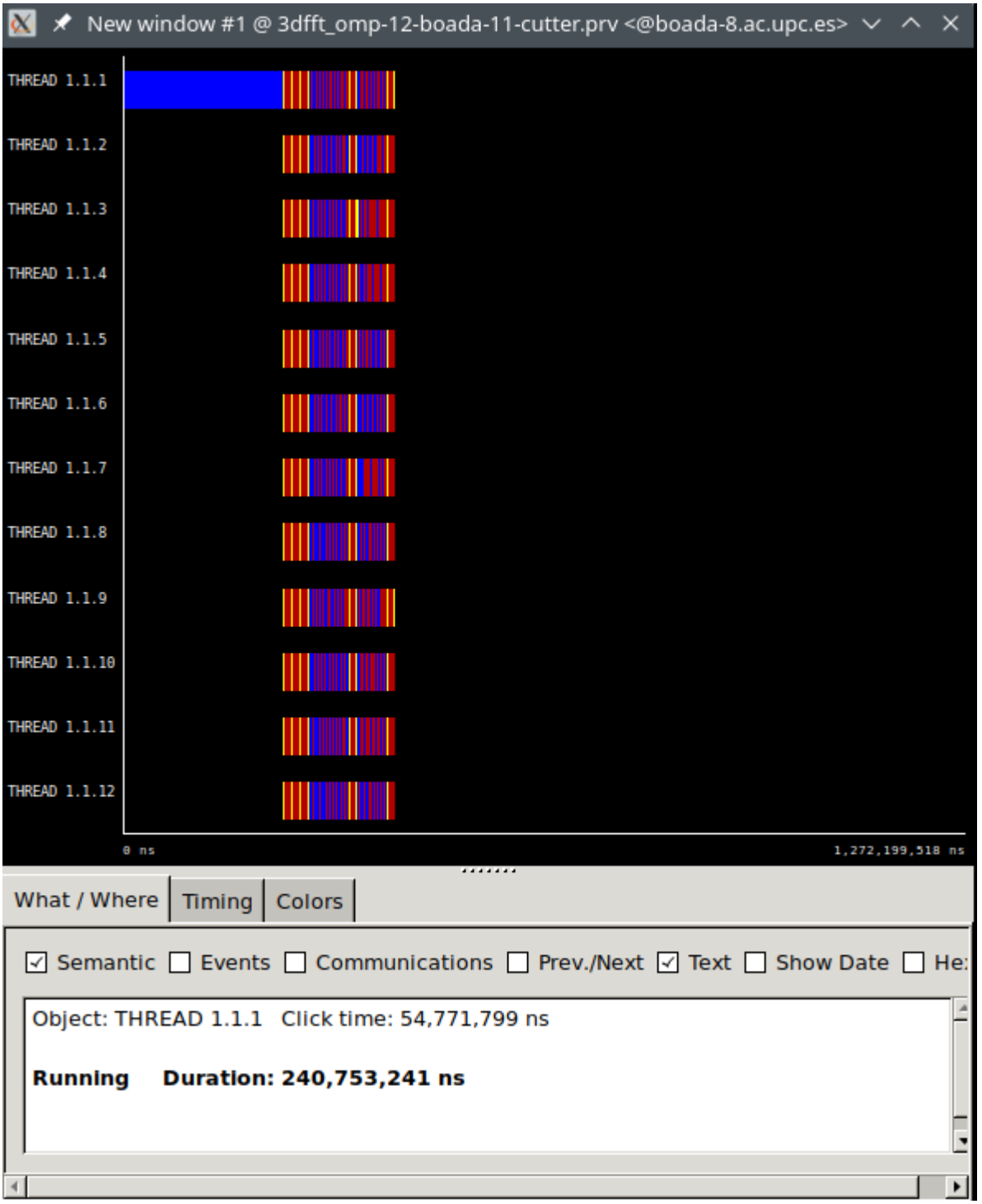
408.257.765 ns.



Si ens fixem en els gràfics en el d'avall, la primera versió del programa, es passa la major part del programa en sincronització, això vol dir que hi ha molt overhead per paral·lelització, en canvi, en la segona versió del programa el color predominant és el blau, que vol dir que la major part del temps està corrent el programa desitjat.

Tanmateix, no sembla que puguem obtenir un temps molt millor augmentant el nombre de threads a partir de 12, ja que com podem veure, i referint-nos a la llei d'Amdahl només podem millorar el tram paral·lelitzable, i més de la meitat del temps que triga el programa actualment és degut a aquesta primera funció que no s'executa en paral·lel.

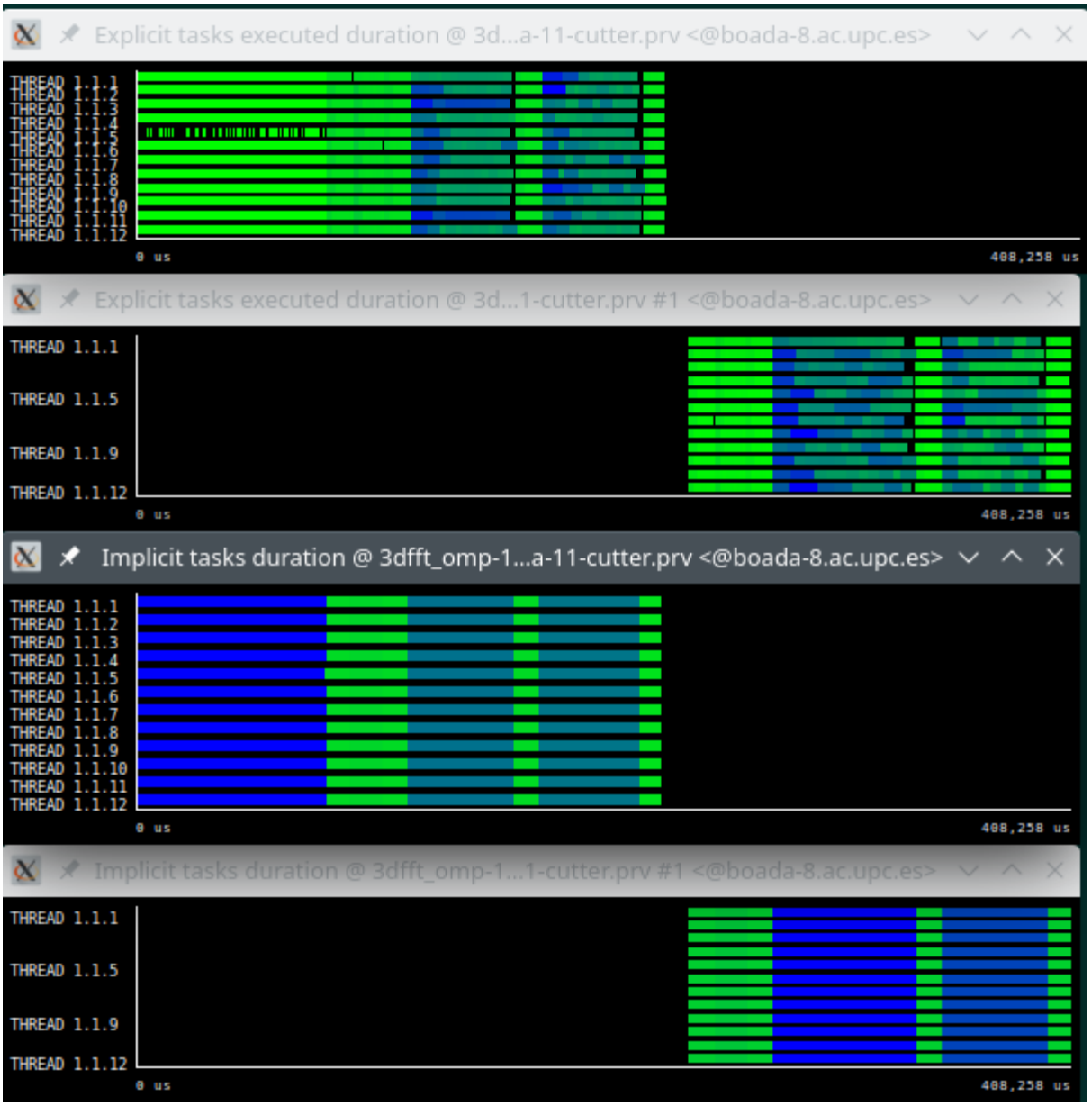
El speedup màxim que es pot aconseguir amb un 82,88% de codi paral·lelitzable és $1/1 - 0.8288 = 5.841$. És a dir 5,8 vegades més ràpid que si no fem servir paral·lelisme. Per tant, aquest valor ens està limitant el speedup que podem assolir.



La funció que ens està limitant el speedup és init_complex_grid amb una duració de 240.753.241. La funció que no està paral·lelitzada és init_complex_grid, això ens està causant un tap d'ampolla al speedup que podem aconseguir.

Respecte a la duració de les tasques implícites podem veure com aquestes tenen menor duració, ja que no han de sincronitzar durant tant de temps, les tasques explícites sembla

que s'ha reduït el seu temps també.



Millorant φ i anàlisis

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.27	0.40	0.25	0.23	0.27
Speedup	1.00	3.13	5.05	5.55	4.73
Efficiency	1.00	0.78	0.63	0.46	0.30

Table 1: Analysis done on Tue Sep 13 05:09:02 PM CEST 2022, par2107

El speedup ha millorat considerablement de nou amb aquesta millora, el valor de paral·lelització ha tornat a augmentar fins al 99,94%.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.94%					
Number of processors	1	4	8	12	16
Global efficiency	99.85%	78.22%	63.17%	46.26%	29.55%
Parallelization strategy efficiency	99.85%	95.16%	92.17%	73.05%	52.89%
Load balancing	100.00%	97.58%	96.23%	95.69%	95.59%
In execution efficiency	99.85%	97.51%	95.77%	76.34%	55.33%
Scalability for computation tasks	100.00%	82.20%	68.54%	63.32%	55.88%
IPC scalability	100.00%	84.21%	74.67%	71.22%	62.82%
Instruction scalability	100.00%	99.80%	99.55%	99.29%	99.03%
Frequency scalability	100.00%	97.80%	92.21%	89.55%	89.83%

Table 2: Analysis done on Tue Sep 13 05:09:02 PM CEST 2022, par2107

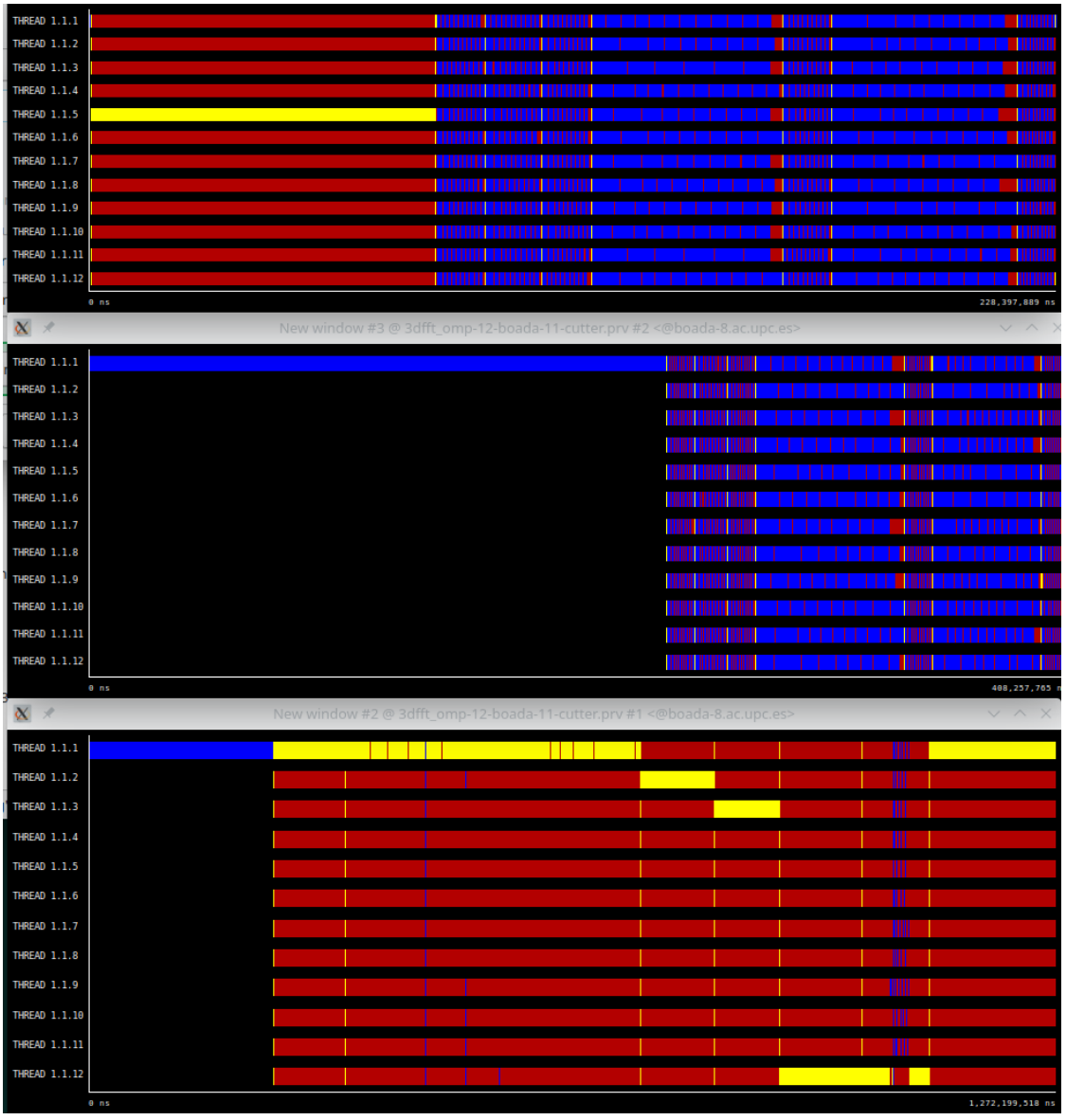
Respecte a la versió inicial i la primera millora l'escalabilitat forta ens ha augmentat prou perquè tingui sentit augmentar el nombre de processadors per a aquesta tasca.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.32	0.78	0.79	1.27	1.47
Speedup	1.00	1.69	1.67	1.04	0.90
Efficiency	1.00	0.42	0.21	0.09	0.06

Table 1: Analysis done on Mon Sep 12 08:14:20 AM CEST 2022, par2107

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	1.27	0.56	0.44	0.41	0.38
Speedup	1.00	2.27	2.88	3.12	3.36
Efficiency	1.00	0.57	0.36	0.26	0.21

Table 1: Analysis done on Tue Sep 13 04:40:15 PM CEST 2022, par2107



Podem veure com a la primera versió (imatge inferior) el temps perdut en sincronització i en overheads és gran, identificat pels colors vermell i groc. En la primera millora (imatge del centre). El fet de tenir tasques més grans, canviant les tasques de granularitat fina per més gruixudes fa que els overheads per sincronització siguin molt menors i el programa pugui fer les tasques que ha de fer. Finalment, en la primera imatge veiem que ja no tenim una tasca inicial no paral·lelitzable que ens limita el speedup del programa.