Treball dirigit PACO:
Recursive exploration with OpenMP - N-Sudoku puzzle

Arnau Roca Grangé, Ixent Cornella

1. Recursive exploration in N-Sudoku

Which is the main data structure used to store the solution that is being explored?

En aquest programa *sudoku.c* ens trobem amb diferents variables a la funció *solve*, però la que guarda la solució que estem buscant és la variable *g*, que és una abreviació de la paraula *grid*, que en anglès vol dir “taula”. Llavors la variable g és un vector que representa una taula on es guarda l’estat de cada sudoku.

Where is it allocated?

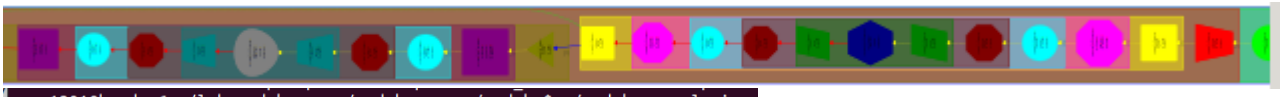
Aquest vector es declara a la funció *solve()*, i inicialitza amb el mètode *new_grid()*. Conseqüentment, el vector segurament estarà alocat al stack, al tractar-se d’una variable local.

How is it initialized?

S’inicialitza mitjançant la funció *read_puzzle()*, la qual llegeix el puzzle que hem de resoldre, i posa un 0 als llocs on hi ha un espai lliure. Als altres espais posa el número corresponent.

What does function all guesses do?

Com diu el seu propi nom, serveix per generar una llista de tots els números candidats que es poden posar a una posició del sudoku.



```
paco1201@boada-6:~/labosudokuarnau/sudokuproves/sudoku$ ./sudoku puzzle.in

Initial puzzle (size 3):
 6 0 0 4 2 0 0 3 0
 0 0 0 8 0 0 0 0 0
 0 0 7 0 3 0 0 0 0
 7 0 2 9 0 0 0 1 0
 0 9 0 0 7 0 6 0 0
 0 0 6 0 0 0 0 0 2
 0 7 0 0 4 0 0 8 0
 0 0 0 7 0 0 0 0 0
 0 6 0 0 9 0 0 0 0

Found 730333 solutions, first one being:
 6 1 5 4 2 7 8 3 9
 2 3 4 8 1 9 5 6 7
 9 8 7 5 3 6 1 2 4
 7 4 2 9 6 5 3 1 8
 1 9 8 3 7 2 6 4 5
 3 5 6 1 8 4 9 7 2
 5 7 9 6 4 1 2 8 3
 8 2 1 7 5 3 4 9 6
 4 6 3 2 9 8 7 5 1

Execution time for Sudoku puzzle: 9.862417s
```

Figura 0 i 1: Graf resultant de la versió inicial i execució inicial

Com podem observar tenim feina, ja que el resultat d'aquest graf es una execució totalment seqüencial. Per tant anem a veure les dependències que causen aquesta execució:

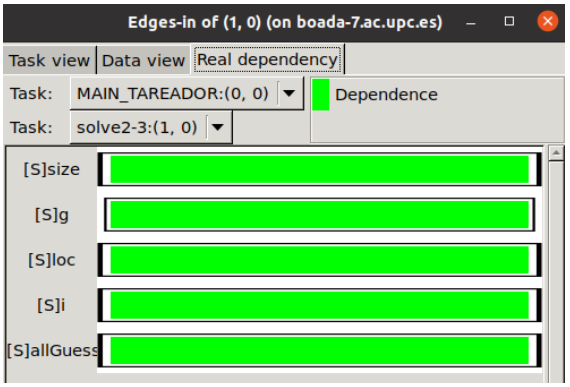


Figura 2: Dependencies "Edges in" al tareador.

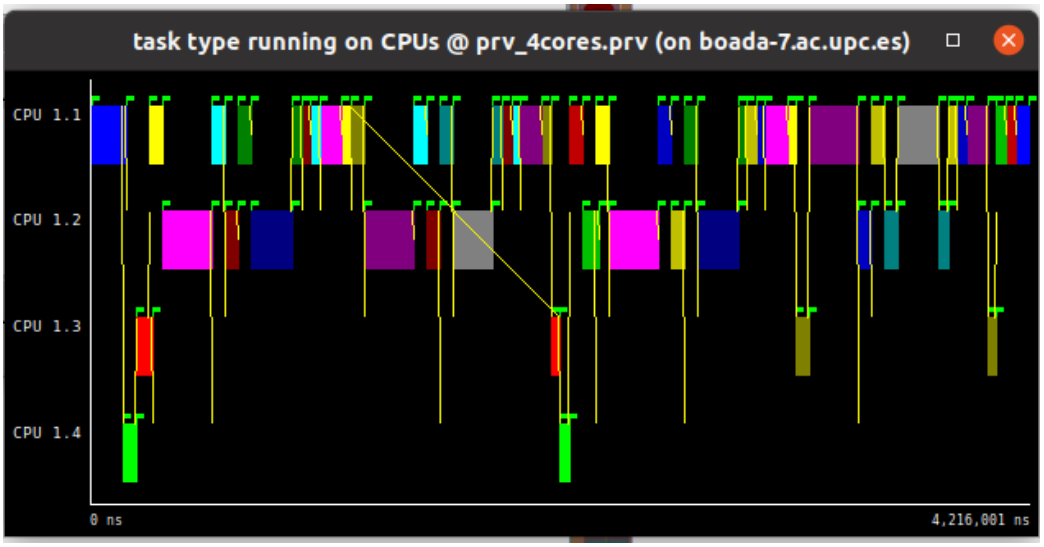
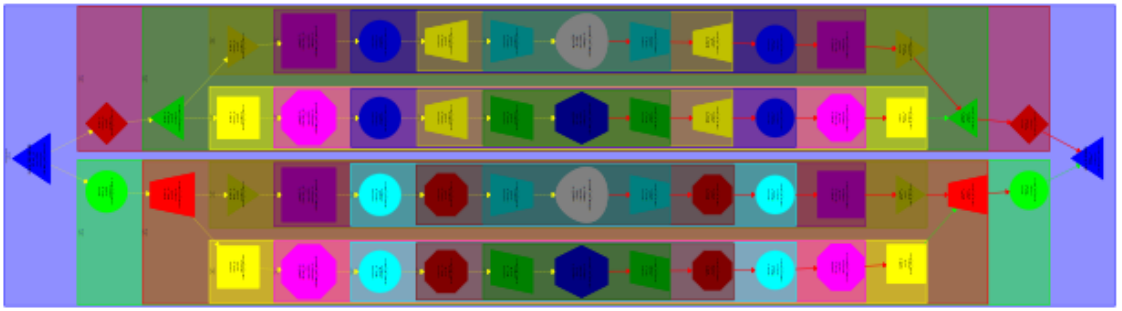


Figura 3: simulació del programa al tareador

Per tal d'aconseguir una gràfica paralelizada el que hem fet és desactivar les variables que ens donen dependències entre iteracions (tareador_enable i tareador_disable). Aquestes son, dins la funció solve, el *num_solutions* i el first_solution. D'aquesta manera ens queda un codi així, potenciant el potential

parallelism:



```
int solve(int size, int* g, int loc)
{
   int i, solved=0;
   int len = size*size*size*size;
   int allGuesses[size*size]; // vector to store all possible solutions for node loc
   int num_guesses; // number of possible solutions for node loc
#if _TAREADOR_

tareador_disable_object(&num_solutions);
tareador_disable_object(&first_solution);
#endif

   /* maximum depth? */
   if (loc == len) {
        num_solutions++;
        if (!first_solution) first_solution = new_grid(size, g); // store first solution found
        return 1;
   }
#if _TAREADOR_
tareador_enable_object(&num_solutions);
tareador_enable_object(&first_solution);
#endif
   /* if this node has a solution (given by puzzle), move to next node */
```

Figures 4 (⬅) i 5(⬆): Codi del sudoku.c i el graf amb potential parallellism

2. Implementation of the OpenMP N-Sudoku

En aquest apartat, hem intentat **paral·lelitzar** el codi de totes maneres, sense èxit. Després de molta estona de intentar-ho i aconseguint **temps molt alts** d'execució (empitjorant l'eficiència del codi inicial), hem decidit seguir endavant fins l'últim punt. Llavors ha sigut quan ens hem adonat del nostre error, si no limitem la **búsqueda en nivells** del programa, aquest estarà explorant nodes **infinitament** sense trobar una solució, per tant la clau és limitar aquesta exploració. Ara ho fem amb una profunditat fixada (4) després en l'últim apartat estudiarem quan és millor tallar la cerca fent l'estudi del cutoff. Així doncs el nostre codi ens quedaria de la següent manera:

```c
int nivells = 4;
unsigned long num_solutions = 0;
int* first_solution = NULL;

int solve(int size, int* g, int loc)
{
   int i, solved=0;
   int len = size*size*size*size;
   int allGuesses[size*size]; // vector to store all possible solutions for node loc
   int num_guesses; // number of possible solutions for node loc

   /* maximum depth? */
   if (loc == len) {
    #pragma omp atomic
    num_solutions++;
    if (!first_solution) first_solution = new_grid(size, g); // store first solution found
        return 1;
   }

   /* if this node has a solution (given by puzzle), move to next node */
   if (g[loc] != 0) {
      solved = solve(size, g, loc+1);
      return solved;
   }

   /* try each number (unique to row, column and square) */
   all_guesses(size, loc, g, allGuesses, &num_guesses);
   /*mirem si estem mes avall dels nivells indicats*/
   if(nivells>loc){
       for (i = 0; i < num_guesses; i++) {
    #if _TAREADOR_
          char buffer[50];
          sprintf(buffer, "solve%d-%d", loc, allGuesses[i]);
          tareador_start_task(buffer);
    #endif
          /*a partir d'aqui explorem recursviament en un taulell nou*/
          int* tauler = new_grid(size,g);
          #pragma omp task shared(solved)
          {
          tauler[loc] = allGuesses[i];
          if (solve(size, tauler, loc+1)) solved = 1;;
          tauler[loc] = 0;

    #if _TAREADOR_
          tareador_end_task(buffer);
    #endif
       }
    }
    #pragma omp taskwait
   }
   else{
      for (i = 0; i < num_guesses; i++) {
       g[loc] = allGuesses[i];
       if(solve(size,g,loc+1)) solved=1;
       g[loc] = 0;
      }
   }

   return solved;
}
```

```c
#endif
   #pragma omp parallel
   #pragma omp single
   solved = solve(size, g, 0);
```

Figures 6 (⬅) i 7(⬆): Codi del sudoku-omp.c (la funció solve sencera i l'únic canvi fet en el main)

Seguidament, pel que fa al codi, cal destacar algunes coses. Podem veure la variable que tenim anomenada *tauler*, el que fa és **crear un nou tauler** igual per tal d'anar explorant els nivells inferiors recursivament fins trobar la solució. També veiem el *task shared* de la variable **solved**, per tal de podem **compartir la variable** entre els diferents threads que tinguem.

```
paco1201@boada-6:~/labosudokuarnau/sudoku$ ./run-omp.sh sudoku-omp puzzle.in 8
make: 'sudoku-omp' is up to date.
/usr/bin/time ./sudoku-omp puzzle.in

Initial puzzle (size 3):
 6 0 0 4 2 0 0 3 0
 0 0 0 8 0 0 0 0 0
 0 0 7 0 3 0 0 0 0
 7 0 2 9 0 0 0 1 0
 0 9 0 0 7 0 6 0 0
 0 0 6 0 0 0 0 0 2
 0 7 0 0 4 0 0 8 0
 0 0 0 7 0 0 0 0 0
 0 6 0 0 9 0 0 0 0

Found 730333 solutions, first one being:
 6 8 9 4 2 1 5 3 7
 1 2 3 8 5 7 4 6 9
 4 5 7 6 3 9 1 2 8
 7 3 2 9 6 4 8 1 5
 5 9 1 2 7 8 6 4 3
 8 4 6 5 1 3 7 9 2
 2 7 5 3 4 6 9 8 1
 9 1 4 7 8 2 3 5 6
 3 6 8 1 9 5 2 7 4

Execution time for Sudoku puzzle: 7.305473s

14.01user 0.67system 0:07.35elapsed 199%CPU (0avgtext+0avgdata 4544maxresident)k
```

Figura 8: execució en run-omp del codi per veure si funciona

Un cop vist que funciona correctament, toca fer l'anàlisi del speedup i la escalabilitat.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.02	0.01	0.01	0.01	0.01
Speedup	1.00	2.83	2.94	2.92	3.10
Efficiency	1.00	0.71	0.37	0.24	0.19

le 1: Analysis done on Mon Jan 2 06:10:58 PM CET 2023, paco1201

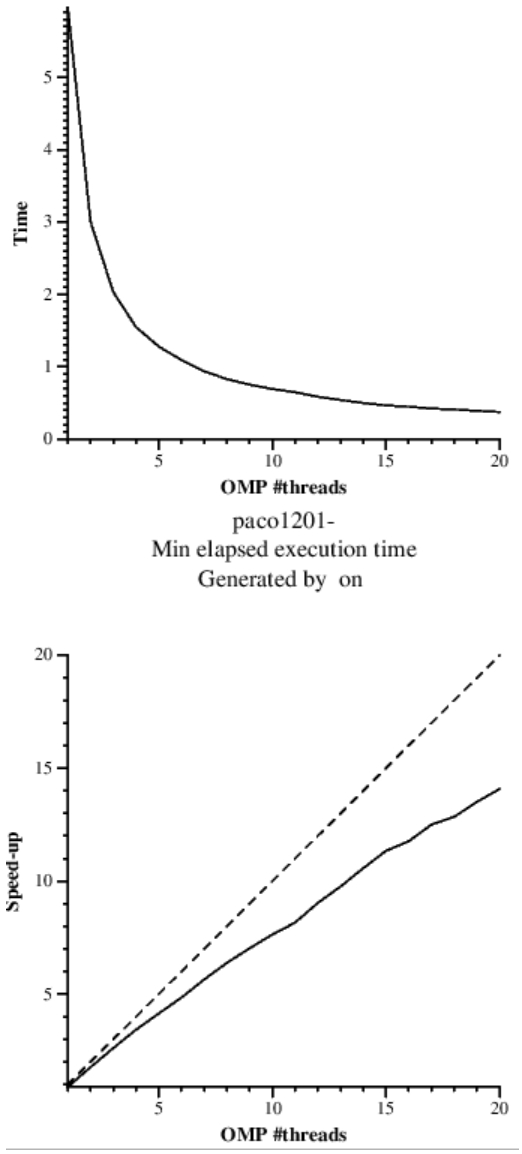
Overview of the Efficiency metrics in parallel fraction, ϕ=97.49%					
Number of processors	1	4	8	12	16
Global efficiency	99.68%	73.39%	37.77%	24.96%	19.96%
Parallelization strategy efficiency	99.68%	76.47%	43.04%	28.88%	23.34%
Load balancing	100.00%	77.52%	43.79%	29.45%	23.90%
In execution efficiency	99.68%	98.65%	98.29%	98.07%	97.66%
Scalability for computation tasks	100.00%	95.98%	87.75%	86.42%	85.49%
IPC scalability	100.00%	94.43%	92.59%	92.50%	91.57%
Instruction scalability	100.00%	100.00%	99.98%	99.97%	99.95%
Frequency scalability	100.00%	101.64%	94.79%	93.46%	93.40%

Table 2: Analysis done on Mon Jan 2 06:10:58 PM CET 2023, paco1201

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	12.0	12.0	12.0	12.0	12.0
LB (number of explicit tasks executed)	1.0	0.6	0.5	0.67	0.55
LB (time executing explicit tasks)	1.0	0.78	0.54	0.64	0.55
Time per explicit task (average us)	1914.75	2004.68	2720.85	3639.17	3618.35
Overhead per explicit task (synch %)	0.04	29.77	105.5	149.51	202.65
Overhead per explicit task (sched %)	0.22	0.73	0.56	0.36	0.55
Number of taskwait/taskgroup (total)	4.0	4.0	4.0	4.0	4.0

Table 3: Analysis done on Mon Jan 2 06:10:58 PM CET 2023, paco1201

Figures 9 i 10: Taules del modelfactors



Com podem observar, els resultats que obtenim son molt bons, ja que ens apropem molt a la línea recta del speedup ideal. I com podem veure el temps baixa significativament desde l'execució amb un thread fins a més threads. Això és degut a que la casuística que hem pensat sobre els nivells que ha de baixar fins a començar l'execució seqüencial és molt adequada, parallelitza molt bé el codi.

Figura 11: Gràfiques de escalabilitat i speed up

3. Parallelization Analysis of N-Sudoku

En aquest apartat ens demanen un anàlisi més exhaustiu dels resultats obtinguts. Executarem el strong extrae tal com ens demanen, per a posteriorment obtenir aquestes dues taules:

Overview of the Efficiency metrics in parallel fraction, ϕ=97.49%					
Number of processors	1	4	8	12	16
Global efficiency	99.68%	73.39%	37.77%	24.96%	19.96%
Parallelization strategy efficiency	99.68%	76.47%	43.04%	28.88%	23.34%
Load balancing	100.00%	77.52%	43.79%	29.45%	23.90%
In execution efficiency	99.68%	98.65%	98.29%	98.07%	97.66%
Scalability for computation tasks	100.00%	95.98%	87.75%	86.42%	85.49%
IPC scalability	100.00%	94.43%	92.59%	92.50%	91.57%
Instruction scalability	100.00%	100.00%	99.98%	99.97%	99.95%
Frequency scalability	100.00%	101.64%	94.79%	93.46%	93.40%

Table 2: Analysis done on Mon Jan 2 06:10:58 PM CET 2023, paco1201

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	12.0	12.0	12.0	12.0	12.0
LB (number of explicit tasks executed)	1.0	0.6	0.5	0.67	0.55
LB (time executing explicit tasks)	1.0	0.78	0.54	0.64	0.55
Time per explicit task (average us)	1914.75	2004.68	2720.85	3639.17	3618.35
Overhead per explicit task (synch %)	0.04	29.77	105.5	149.51	202.65
Overhead per explicit task (sched %)	0.22	0.73	0.56	0.36	0.55
Number of taskwait/taskgroup (total)	4.0	4.0	4.0	4.0	4.0

Table 3: Analysis done on Mon Jan 2 06:10:58 PM CET 2023, paco1201

Figures 12 i 13: taules del modelfactor.

Com podem observar a les taules generades, el nombre de tasques creades es 12, independentment del número de threads que s'utilitzin.

Pel que fa al temps de l'execució per thread, obtenim els següents resultats (en milisegons):

- Amb 1 thread → 1,914 ms
- Amb 4 threads → 2,004 ms
- Amb 8 threads → 2,720 ms

- Amb 12 threads → 3,639 ms
- Amb 16 threads → 3,618 ms

Es pot comprovar a les taules adjuntes que no hi ha millora en temps amb més threads; de fet és al contrari, empitjora el temps d'execució amb més threads, tal i com deia l'enunciat (*"You should have observed slowdown on the execution of your parallel program for more than one thread"*). Per tant, assumim que el resultat és correcte.

Aquí hi ha altres figures per a l'execució amb 12 threads:

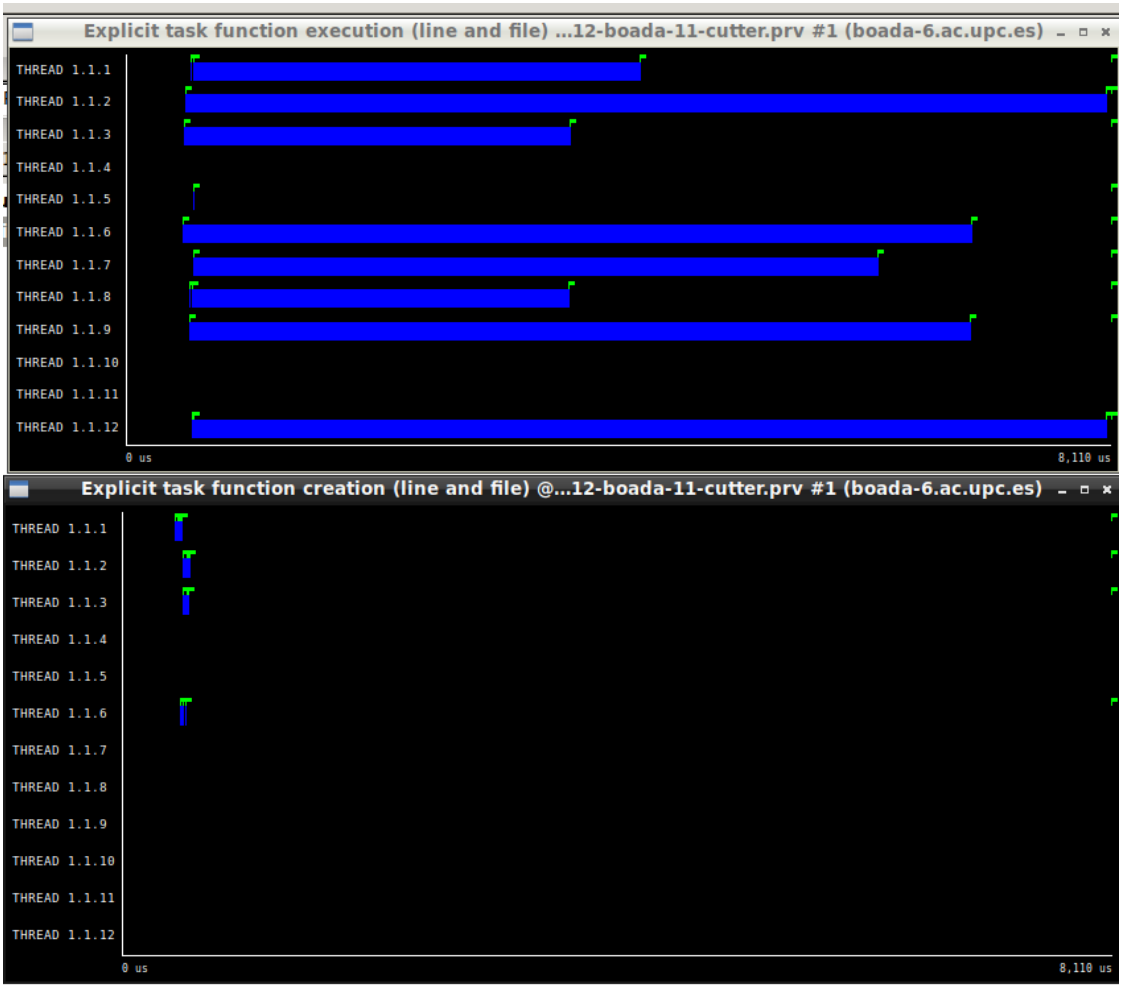


Figura 14: execució i creació de tasques.

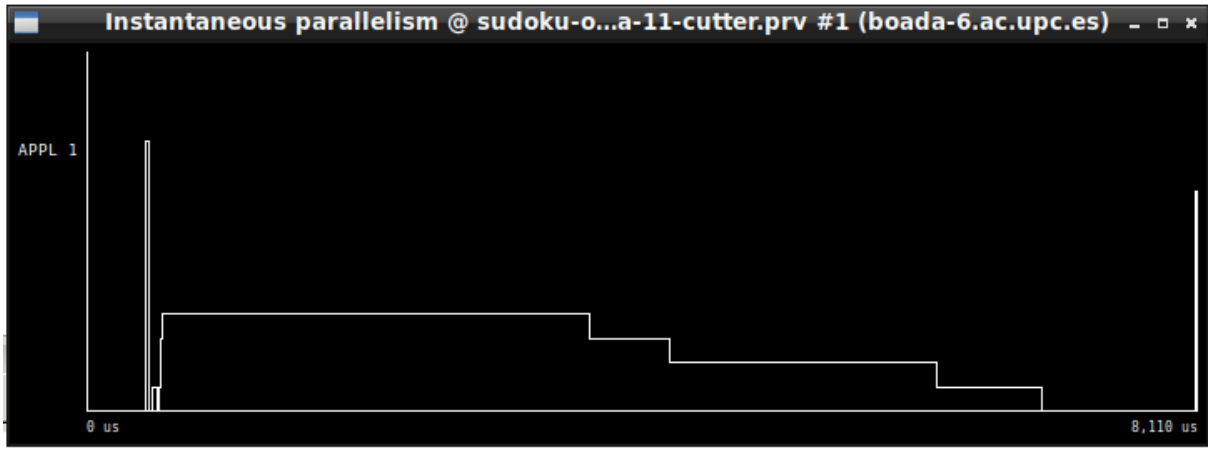


Figura 15: paral·lelisme instantani (12 threads).

4. Optimization of the OpenMP N-Sudoku

Ara com ja hem anticipat abans, en el segon punt haurem de buscar el valor més adequat de nivells que hem de baixar. Per fer-ho modifiquem el codi per tal que agafi per paràmetre el número que li indiquem per baixar. El codi ens quedaria de la següent manera:

```
if (argc < 2) {
     fprintf(stderr, "Usage: %s <puzzle_filename> \n or <puzzle_filename> <-c> <cutoff value>\n", argv[0]);
     return(0);
}
      if(argv[2]){
            nivells=atoi(argv[3]);
     }
     else {
            fprintf(stderr, "Usage: %s <puzzle_filename> \n or <puzzle_filename> <-c> <cutoff value>\n", argv[0]);
            return(0);
     }
```

Figura 16: Part del codi modificada.

Com podem veure, l'únic canvi es troba en aquest lloc del codi, també a dalt de tot, canviaria que no inicialitzem la variable nivells. Fet això veiem quin és el millor cutoff:

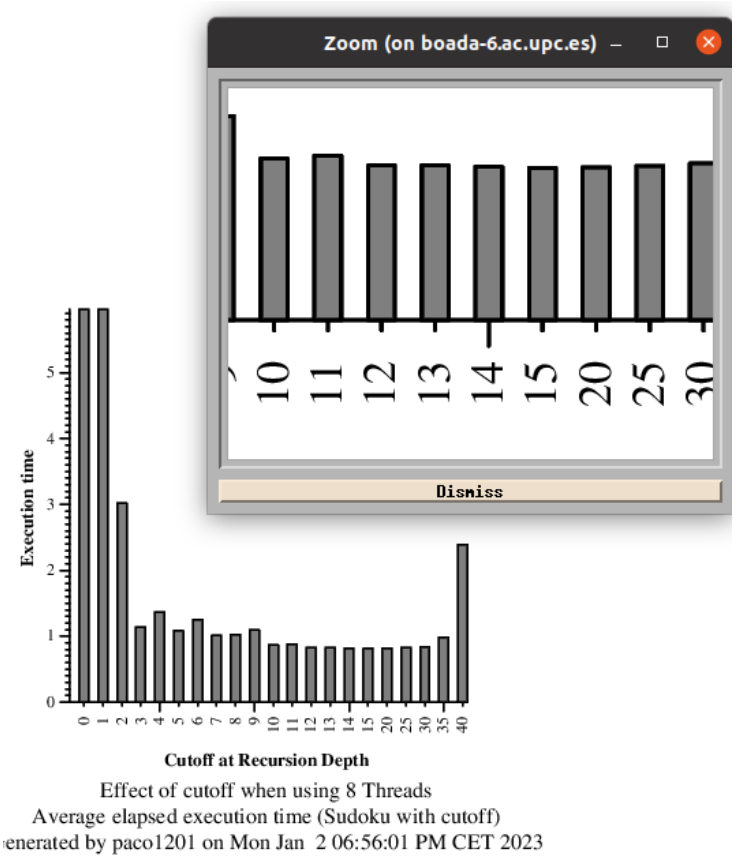


Figura 17: Resultat del script del cutoff.

Com veiem en la anterior figura, hi ha un empat tècnic, nosaltres ens quedarem amb el 13, per provar-ho i repetirem tot el que hem vist al apartat 3, amb poques millores, ja que ja estava molt bé.

```
paco1201@boada-6:~/labosudokuarnau/sudoku$ ./sudoku-omp puzzle.in

Initial puzzle (size 3):
 6 0 0 4 2 0 0 3 0
 0 0 0 8 0 0 0 0 0
 0 0 7 0 3 0 0 0 0
 7 0 2 9 0 0 0 1 0
 0 9 0 0 7 0 6 0 0
 0 0 6 0 0 0 0 0 2
 0 7 0 0 4 0 0 8 0
 0 0 0 7 0 0 0 0 0
 0 6 0 0 9 0 0 0 0

Found 730333 solutions, first one being:
 6 1 9 4 2 7 8 3 5
 5 4 3 8 1 6 2 7 9
 2 8 7 5 3 9 1 4 6
 7 3 2 9 6 5 4 1 8
 1 9 8 2 7 4 6 5 3
 4 5 6 1 8 3 7 9 2
 3 7 5 6 4 2 9 8 1
 9 2 1 7 5 8 3 6 4
 8 6 4 3 9 1 5 2 7

Execution time for Sudoku puzzle: 5.542662s

paco1201@boada-6:~/labosudokuarnau/sudoku$
```

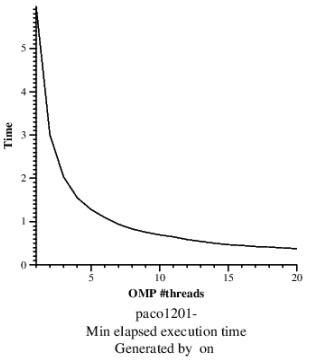
```
  GNU nano 6.2                  time-sudoku-puzzle.in-8-boada-11
5.31user 0.00system 0:05.33elapsed 99%CPU (0avgtext+0avgdata 2296maxresident)k
0inputs+8outputs (0major+101minor)pagefaults 0swaps
```

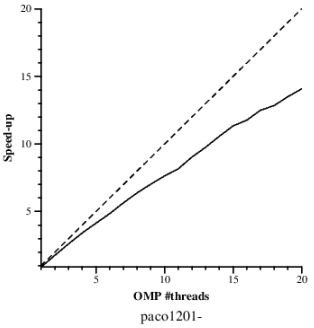
Overview of the Efficiency metrics in parallel fraction, ϕ=96.52%					
Number of processors	1	4	8	12	16
Global efficiency	99.40%	83.48%	69.99%	59.97%	31.09%
Parallelization strategy efficiency	99.40%	86.39%	80.01%	70.30%	37.42%
Load balancing	100.00%	90.60%	85.93%	80.34%	45.04%
In execution efficiency	99.40%	95.35%	93.11%	87.50%	83.09%
Scalability for computation tasks	100.00%	96.63%	87.48%	85.30%	83.08%
IPC scalability	100.00%	97.68%	93.54%	93.11%	91.06%
Instruction scalability	100.00%	100.12%	100.10%	100.08%	100.07%
Frequency scalability	100.00%	98.80%	93.43%	91.54%	91.17%

Table 2: Analysis done on Mon Jan 2 12:41:39 PM CET 2023, paco1201

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	150.0	150.0	150.0	150.0	150.0
LB (number of explicit tasks executed)	1.0	0.68	0.62	0.62	0.67
LB (time executing explicit tasks)	1.0	0.91	0.87	0.81	0.49
Time per explicit task (average us)	151.61	160.9	181.01	192.51	219.61
Overhead per explicit task (synch %)	0.24	13.59	20.21	32.98	127.25
Overhead per explicit task (sched %)	0.33	1.69	3.46	5.73	11.59
Number of taskwait/taskgroup (total)	88.0	88.0	88.0	88.0	88.0

Table 3: Analysis done on Mon Jan 2 12:41:39 PM CET 2023, paco1201





Figures 18, 19, 20, 21: Resultat de l'execució amb cutoff 13, gràfiques i taules de speedup.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.02	0.01	0.00	0.00	0.01
Speedup	1.00	3.13	5.19	6.32	4.54
Efficiency	1.00	0.78	0.65	0.53	0.28

Table 1: Analysis done on Mon Jan 2 12:41:39 PM CET 2023, paco1201

Com s'observa, el temps d'execució millora significativament, tal com alguns dels valors de les taules, però el que més destaca seria el temps. A continuació tenim les captures del paraver, que no difereixen molt del resultat que hem obtingut i comentat abans i per tant no cal dir res.

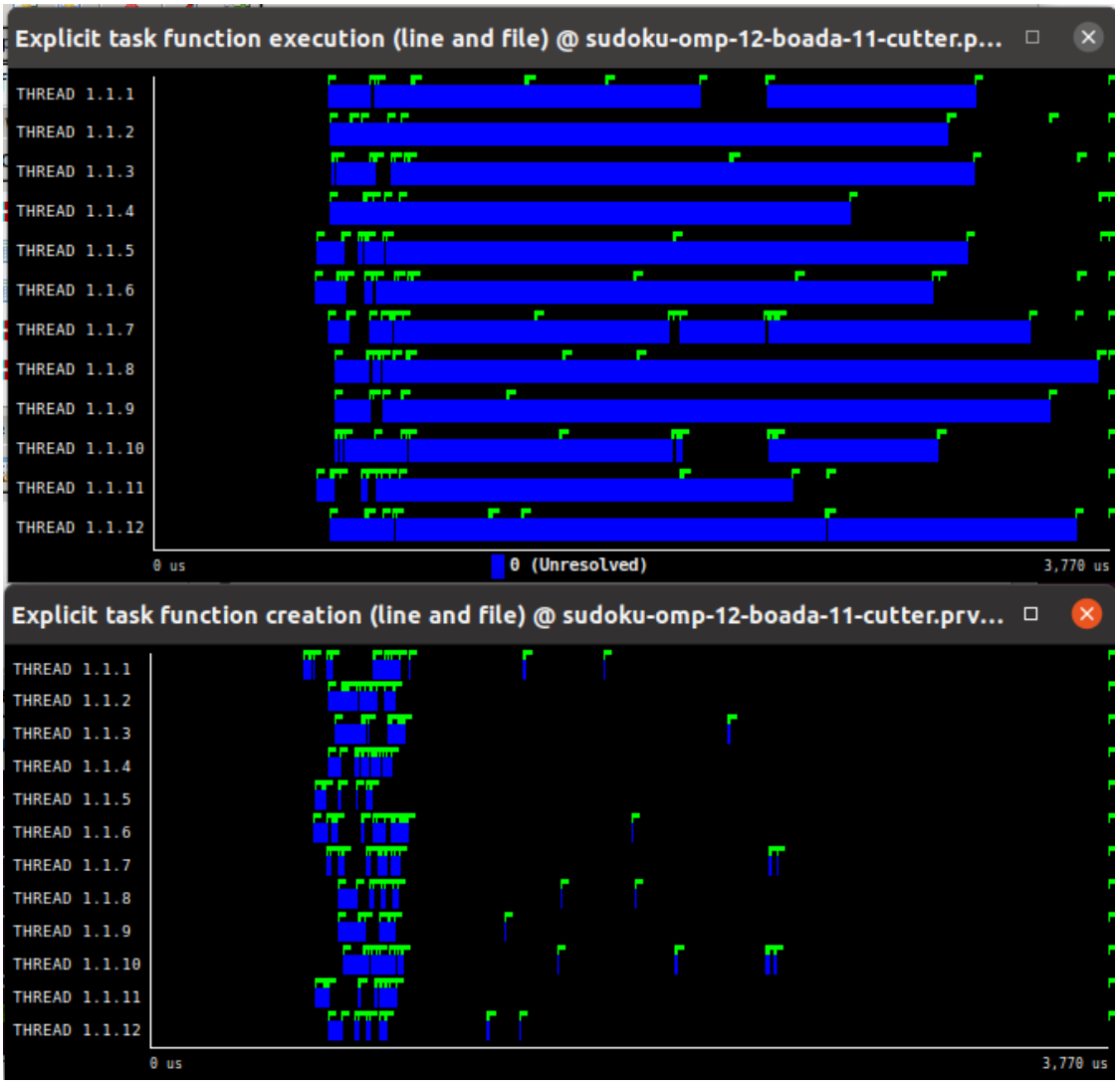


Figura 22: execució i creació de tasques.

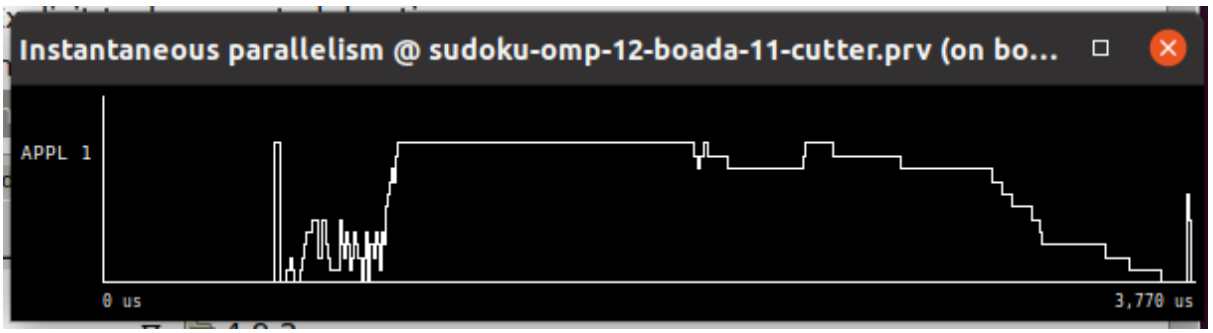


Figura 23: paral·lelisme instantani de l'execució

Useful+OpenMP+OpenMP tasking+PAPI counte

Histogram task instantiation @ sudoku-o...

	0 (Unresolved)
THREAD 1.1.1	16
THREAD 1.1.2	11
THREAD 1.1.3	13
THREAD 1.1.4	11
THREAD 1.1.5	7
THREAD 1.1.6	18
THREAD 1.1.7	15
THREAD 1.1.8	10
THREAD 1.1.9	7
THREAD 1.1.10	18
THREAD 1.1.11	13
THREAD 1.1.12	11
Total	150
Average	12.50
Maximum	18
Minimum	7
StDev	3.57
Avg/Max	0.69

Histogram task execution @ sudoku-o...

	0 (Unresolved)
THREAD 1.1.1	12
THREAD 1.1.2	6
THREAD 1.1.3	12
THREAD 1.1.4	7
THREAD 1.1.5	11
THREAD 1.1.6	16
THREAD 1.1.7	19
THREAD 1.1.8	15
THREAD 1.1.9	8
THREAD 1.1.10	20
THREAD 1.1.11	11
THREAD 1.1.12	13
Total	150
Average	12.50
Maximum	20
Minimum	6
StDev	4.23
Avg/Max	0.62

Figura 24: nombre de tasques creades i executades.

5. Conclusions

En conclusió, el programa inicial ens proporcionava un temps d'execució de gairebé 10 segons aproximadament, però el resultat de la paral·lelització ens ha aportat una millora d'aquest resultat, fins arribar als 5 segons d'execució, per tant, el projecte es pot calificar objectivament com a un èxit rotund.

A nivell personal, aquest petit treball dirigit ens ha servit per aplicar els coneixements de PACO a un problema que podria ser real, a més de fer-nos veure les dificultats que hi ha a l'hora de paralelitzar un programa, ja que desde l'inici del treball ens hem trobat amb problemes, tot i que eventualment les hem pogut resoldre fins a poder completar el treball amb solvència.