Parallelism

Lab 4: Branch and bound with OpenMP: N-queens puzzle

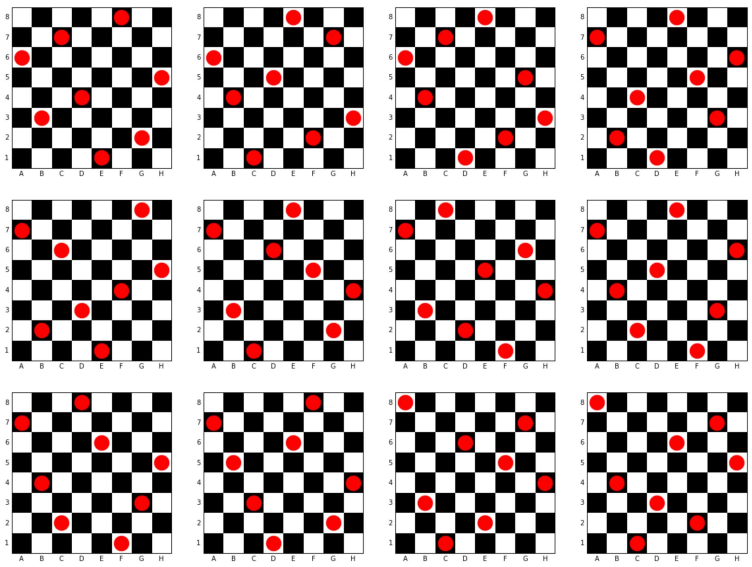
Marian Danci & David Valero
Group 42, par4207
Fall 2018-19

Index

Introduction

For this Lab4 we have worked with a Branch and Bound technique in a backtracking algorithm, specifically with N-queens algorithm. This technique is widely used for speeding up backtracking algorithms. In the beginning you have a recursive algorithm that tries to build a solution part by part, and when it ends, then it has either built a solution or it needs to go back (backtrack) and try with different values for some of the parts. To check if the solution built is a valid solution is done at the deepest level of recursion, when all parts have been picked out. However, after building only a partial solution the algorithm can decide that there is no need to go any deeper because it is heading into a dead end.
We will work with the Nqueens puzzle, which it is the problem of placing N chess queens on an N × N chessboard so that no two queens attack each other. Therefore, a solution does not have two queens share the same diagonal, row or column.

During the first session, we have been working in understanding this type of algorithms and study how we could paralyse it with Tareador. After the analysis, we have paralysed the algorithm with OpenMP. The main difference between this new Laboratory and all before is that we focus on how we can do the shared-memory parallelization.



Picture1: Example of the all solution with a size board of 8.

Session 1: Understanding the potential parallelism in N-queens

As we explain before, we try to understand the potential parallelism in the N-queens algorithm with *Tareador*.
First of all, we focus on the analysis of how the algorithm work. The idea of the N-queens is how we can put N queens in a chess board where no one can kill another queen.
As we saw in the 'nqueens.c' code, the idea of brunch and bound technique is to go ramifying the search tree.

So, as we can see, the algorithm starts at point (0, 0) and with the 'ok' function, is determined if the queen can be put there. If is it possible, it calls the 'nqueens' function with the iterator 'j' parameter plus one, this allows to branch out the algorithm, to be able to obtain all the possible solutions. If not, he continues iterating through a 'for'.
As we can deduce, the algorithm goes through the matrix of positions, and when it arrives at the last position (if the matrix has m x m position, the last position is (m - 1, m - 1)), it enters in the first 'if' of the function and puts the solution in the heap and adds one to the solution counter, 'sol_count' (which at the beginning of the algorithm was equal to zero).

To confirm the good the proper functioning of the code, we compile the sequential version of the program using "make nqueens-seq" and we execute the binary using "./nqueens-seq -n12".
We get this execution report:

```
Parameters:
  board size:               12
  recursion cutoff level:   8

one solution:  0  2  4  7  9 11  5 10  1  6  8  3
number of solutions: 14200
Solution Count Time is 0.956412 seconds
```

After that, we open the 'nqueens.c' source code and we try to understand Tareador instrumentation provided.

We observed that there are some '`#ifndef _TAREADOR_`' clauses, to compile it with or without the *Tareador* clauses with the same code. You can see the structure in Code1:

```
#ifndef _TAREADOR_
     aquesta part nomes forma part del programa si es compila
en la version -tar
#endif
```

Code1: Structure of a #ifdef clause..

Focusing on the main function, it is put a '`tareador_ON()`' before the nqueen function call and a '`tareador_OFF()`' after that, to initialize the Tareador. You can see a fragment of the code in Code2:

```
int main(int argc, char *argv[])
{
     int i;

     process_args(argc, argv, &size, &cutoff);

     a = alloca(size * sizeof(char));

     printf( "Parameters:\n" );
     printf( "  board size:              %d\n", size );
     printf( "  recursion cutoff level:  %d\n", cutoff );
     printf( "\n" ) ;
     fflush( stdout );

#ifndef _TAREADOR_
     double stamp;
#endif

#ifdef _TAREADOR_
     tareador_ON ();
#else
     START_COUNT_TIME;
#endif

     nqueens(size, 0, a);

#ifdef _TAREADOR_
     tareador_OFF ();
#else
     STOP_COUNT_TIME;
#endif
```

Code2: Fragment of the Main function where we put the Tareador clauses.

Then, at the 'nqueens' function, it is put a '`tareador_start_task("Solution")`' before and '`tareador_end_task("Solution")`' after the first if clause, then inside the else clause, we put another '`tareador_start_task(stringMessage)`' and '`tareador_end_task(stringMessage)`' in the 'if' clause.

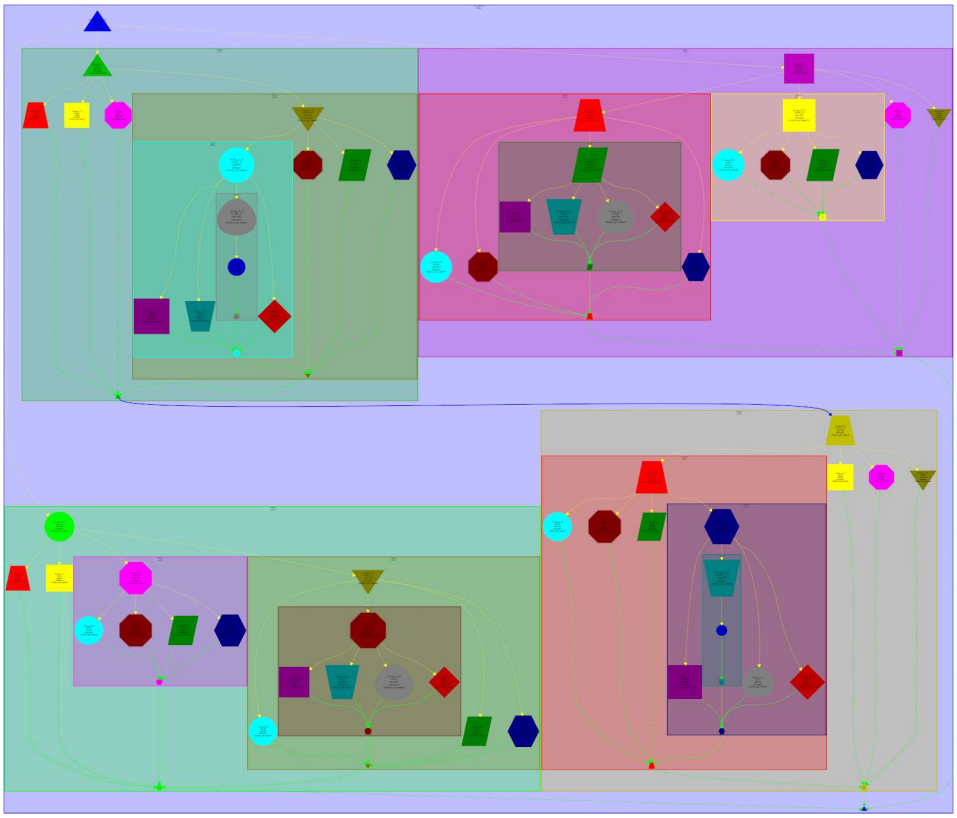
These clauses generates a task at every if clause, one when a solution is found and every time the ‘ok’ function is called, highlight that to show better the recursion clauses, every task has a different name. You can see in the Code3:

```
void nqueens(int n, int j, char *a) {
     int i;

     if (n == j) {
#ifdef _TAREADOR_
          tareador_start_task("Solution");
#endif
          /* put good solution in heap. */
          if( sol == NULL ) {
                    sol = malloc(n * sizeof(char));
                    memcpy(sol, a, n * sizeof(char));
          }
          sol_count += 1;
#ifdef _TAREADOR_
          tareador_end_task("Solution");
#endif
     } else {
          /* try each possible position for queen <j> */
          for ( i=0 ; i < n ; i++ ) {
#ifdef _TAREADOR_
               sprintf(stringMessage,"Trying [%d,%d]",j,i);
               tareador_start_task(stringMessage);
#endif
               a[j] = (char) i;
               if (ok(j + 1, a)) {
                    nqueens(n, j + 1, a);
               }
#ifdef _TAREADOR_
               tareador_end_task(stringMessage);
#endif
          }
     }
}
```

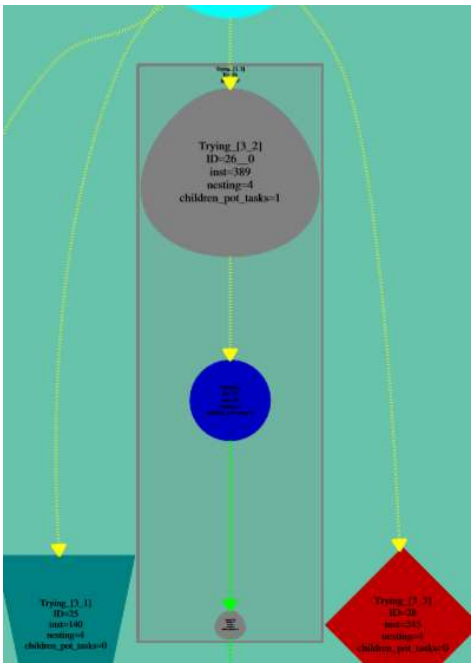
Code3: nqueens function with Tareador clauses.

Then, we compile the Tareador instrumented version of the program using "make nqueens-tar" and we execute the binary using the ‘run-tareador.sh’ script, we have obtained the picture1:



Picture2: Task dependences graph of n-queens.

As you can see, there is one block per task and only two solutions in a 4x4 board.
After calling for the first time the 'nqueens' function, it creates four big tasks (the four big blocks) one for each column of the 0 row.
After that, it follows the same system until finding a solution, where the structure changes a little, as you can see in the picture below.



Picture3: Fragment of picture1, one solution task.

As we can see, the dependencies are between one level and the other and each iteration of the ‘for’ loop can be run in parallel.
Furthermore, at the data level, the dependencies are between the vectors that represent the board.
To solved that problem of dependencies, one different vector for each task is created, generating a dynamic allocation of memory, by the vectors.

Session 2: Shared-memory parallelization

In this second section of the laboratory assignment, we have parallelized the original sequential code using OpenMP tasks to explore solutions for the possible n positions of a queen in each column.
Before it, as we see at the last section, there are dependencies between [0,2] and [0,1] and because of having a vector holding the solutions, the code needs to reserve memory and copy it when it finds one. As there are assignments to shared variables, this produces a dependence when we save it. To solve it, we can use a 'critical' or 'atomic' clause and we had to use the following functions:

- 'alloca(size)' to allocate size bytes in memory that is automatically freed
- 'memcpy(dest, src, size)' to copy size bytes from memory area pointed by src to memory area pointed by dest.

With this functions, we avoid sharing the vector 'a' among the tasks.

To parallelize we considered the following factors. First of all, we added in the main function a '#pragma omp parallel' and '#pragma omp single' over the 'nqueens' call, after this we only applied changes in the 'nqueens' function. At first, we create a task each iteration of the 'for' loop and due to the task do not have parent all variables are firstprivate, so we do not have to specific them.
To capture the pointed data we copied it to the memory. In our version, we protect the shared variable 'sol_count' with a '`#pragma omp atomic`'.
You can see the code below.

```
void nqueens(int n, int j, char *a) {
     int i;
     if (n == j) {
      /* put good solution in heap. */
          if( sol == NULL ) {
                    sol = malloc(n * sizeof(char));
                    memcpy(sol, a, n * sizeof(char));
          }
          #pragma omp atomic
          sol_count += 1;
     } else {
          /* try each possible position for queen <j> */
          for ( i=0 ; i < n ; i++ ) {
              #pragma omp task
              {
                   char *b = alloca (n * sizeof(char));
                   memcpy(b, a, n * sizeof(char));
                   b[j] = (char) i;
                   if (ok(j + 1, b)) {
                       nqueens(n, j + 1, b);
                       }
                   }
          }
     }
     #pragma omp taskwait
}
```

Code4: nqueens function with pragma clauses.

Then, in the else clause, before the for, we put a '`#pragma omp task`' (no '#pragma omp for', therefore no reduction can be made), grouping the if clause. After it, we edit this peace of code:

```
char *b = alloca(n * sizeof(char));
memcpy(b, a, n * sizeof(char));
b[j] = (char) i;
```

These guarantee the good reserve of memory. With the 'alloca' and 'memcpy' function it creates a copy and reserve memory dynamically for the vector. When it finish the task, it frees it automatically. Finally, a '`#pragma omp taskwait`' is put to guarantee the good trade of dependencies.

After compiling our parallel version, we execute the binaries generated using the 'run-omp.sh' (interactively) and the 'submit-omp.sh' (through execution queues) script in order to check correctness and measure parallel execution time.
Note that despite putting in the script the recursion cutoff level, no cutoff variable is used here. We get this results:

```
par4207@boada-1:~/lab4$ ./run-omp.sh nqueens-omp 12 8 8
make: 'nqueens-omp' is up to date.
Parameters:
  board size:              12
  recursion cutoff level:  8

one solution:  1 11  9  7 10  3  0  2  5  8  6  4

number of solutions: 14200

Solution Count Time is 1.120086 seconds
4.58user 4.35system 0:01.12elapsed 793%CPU (0avgtext+0avgdata
4624maxresident)k
0inputs+0outputs (0major+832minor)pagefaults 0swaps
```

Result1: Results of the run-omp.sh script.

```
par4207@boada-1:~/lab4$ qsub -l execution submit-omp.sh nqueens-omp 12 8 8
4.96user 3.99system 0:01.17elapsed 762%CPU (0avgtext+0avgdata
4624maxresident)k
160inputs+8outputs (1major+830minor)pagefaults 0swaps
```

Result2: Result of the submit-omp.sh script.

As we can see in both scripts, despite the results are correct and the code takes to execute 1.12 seconds, the code generates an excessive task creation overheads.
In order to avoid it, we need to control the recursion level up when we create tasks using a cut-off variable.

To do it, we change the original pragma task clause of the else part to a '`#pragma omp task final(depth >= CUTOFF) mergeable`', where 'depth' is a parameter of the function the level of depth of it. You can see the final result in the code below, we also added 'mergeable' to improve the performance.

```
void nqueens(int n, int j, char *a, int depth) {
     int i;
     if (n == j) {
      /* put good solution in heap. */
          if( sol == NULL ) {
                    sol = malloc(n * sizeof(char));
                    memcpy(sol, a, n * sizeof(char));
          }
          #pragma omp atomic
          sol_count += 1;
     } else {
          /* try each possible position for queen <j> */
          for ( i=0 ; i < n ; i++ ) {
              #pragma omp task final(depth >= CUTOFF) mergeable
              {
                   char *b = alloca (n * sizeof(char));
                   memcpy(b, a, n * sizeof(char));
                   b[j] = (char) i;
                   if (ok(j + 1, b)) {
                       nqueens(n, j + 1, b, ++depth);
                       }
                   }
          }
     }
     #pragma omp taskwait
}
```

Code5: nqueens function with pragma clauses and cut-off.

We compile again our parallel version and we execute the binaries generated using the 'submit-omp.sh' (through execution queues) script in order to check correctness and measure parallel execution time. We obtain this results:

```
par4207@boada-1:~/lab4$ qsub -l execution submit-omp.sh nqueens-omp 12 2 8
3.60user 1.88system 0:00.65elapsed 771%CPU (0avgtext+0avgdata
4552maxresident)k
160inputs+8outputs (1major+457minor)pagefaults 0swaps

par4207@boada-1:~/lab4$ qsub -l execution submit-omp.sh nqueens-omp 12 8 8
3.54user 1.34system 0:00.55elapsed 725%CPU (0avgtext+0avgdata
4500maxresident)k
0inputs+8outputs (0major+453minor)pagefaults 0swaps
```

```
par4207@boada-1:~/lab4$ qsub -l execution submit-omp.sh nqueens-omp 12 12 8
2.98user 0.75system 0:00.60elapsed 764%CPU (0avgtext+0avgdata
4588maxresident)k
0inputs+8outputs (0major+460minor)pagefaults 0swaps


par4207@boada-1:~/lab4$ qsub -l execution submit-omp.sh nqueens-omp 12 20 8
3.54user 1.84system 0:00.70elapsed 770%CPU (0avgtext+0avgdata
4504maxresident)k
0inputs+16outputs (0major+455minor)pagefaults 0swaps
```

Result3: Result of the submit-omp.sh script with cutoff.

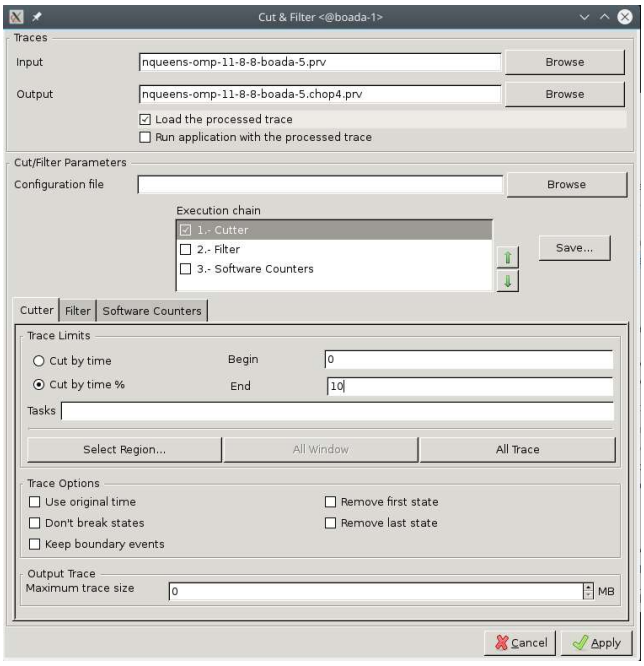
We see that the cutoff level has a great effect in the execution time, reducing the user time of 4.96 to 3.46 (1.5 seconds less), the system time of 3.99 to 2.01 (1.98 seconds less) and the elapsed time of 1.17 to 0.71 (0.46 seconds less).

Furthermore, we analyze our parallel code using Paraver and we obtain that the most appropriate value for the cutoff parameter, depending on the number of threads used to execute the program we see that the best one in our case is eight.

Due to we did not have enough space in the directory we make the trace with a smaller size of the ‘n’, 11 . To submit to the queue we used:

```
qsub -l cuda submit-extrae.sh nqueens-omp 11 8 8
```

Once generated the trace to open the .prv file wi had to cut it in different parts, because of his size. As we can see it in the following image.

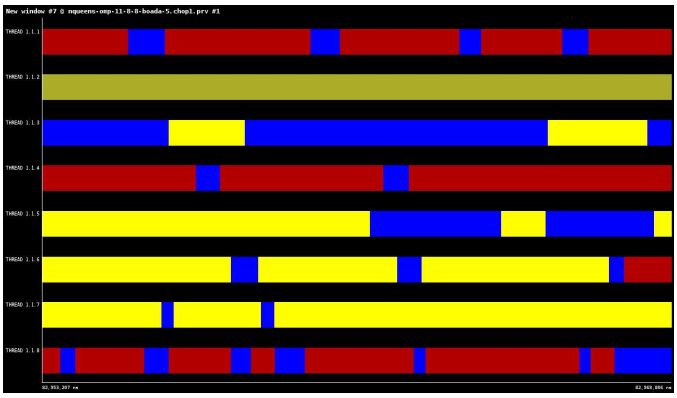


Picture4: Cut&Filter paraver.

We study the trace three parts, to 0%-10%, to 35%-65% and to 90%-%100.

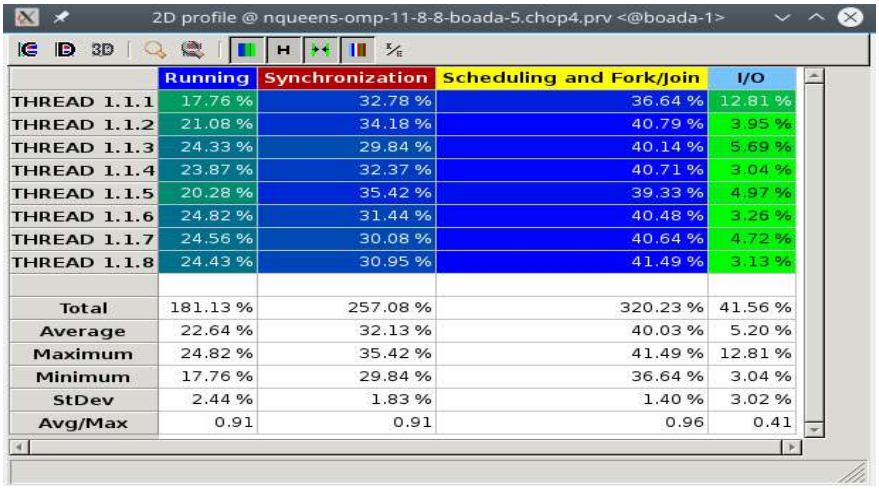
Picture5: Trace of nqueens-omp-11-8-8-boada-5 to 0%-10%, to 35%-65% and 90%-100%.

The only remarkable there is the great amount of sequential time that the program spend doing input and output processes, which are the dark yellows, meaning these parts are not interrupted at any time. As we can see in the following picture.



Picture6: Traceof nqueens-omp-11-8-8-boada-5 to 35%-65% zoom in i/o part.

Afert that we loaded a configuration file to saw in what each thread spend its time. And we can see there are a great overhead, this may be because of the lack of optimization of our code.



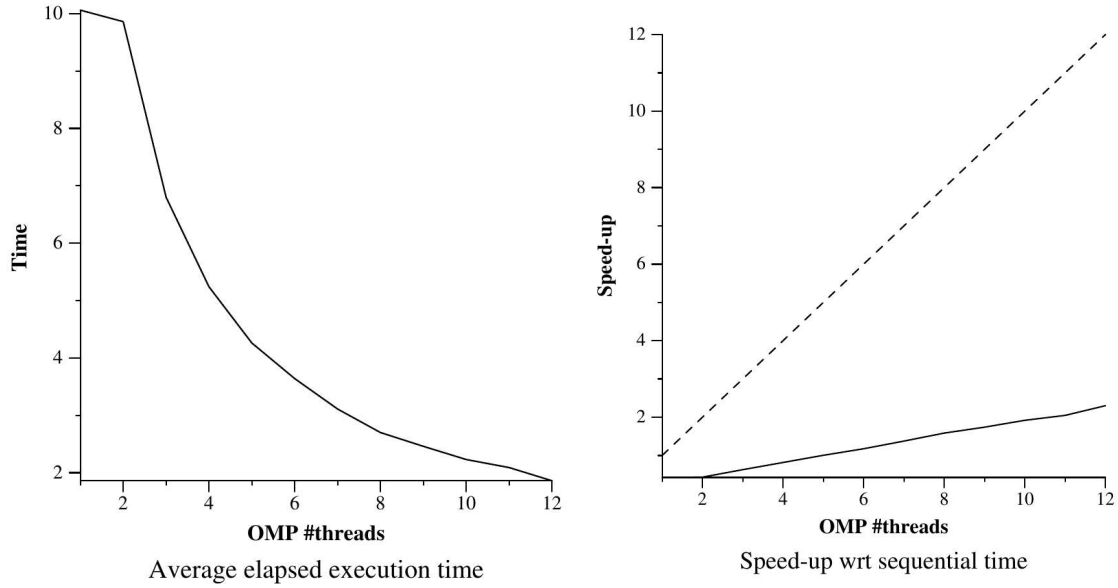
	Running	Synchronization	Scheduling and Fork/Join	I/O
THREAD 1.1.1	17.76 %	32.78 %	36.64 %	12.81 %
THREAD 1.1.2	21.08 %	34.18 %	40.79 %	3.95 %
THREAD 1.1.3	24.33 %	29.84 %	40.14 %	5.69 %
THREAD 1.1.4	23.87 %	32.37 %	40.71 %	3.04 %
THREAD 1.1.5	20.28 %	35.42 %	39.33 %	4.97 %
THREAD 1.1.6	24.82 %	31.44 %	40.48 %	3.26 %
THREAD 1.1.7	24.56 %	30.08 %	40.64 %	4.72 %
THREAD 1.1.8	24.43 %	30.95 %	41.49 %	3.13 %
Total	181.13 %	257.08 %	320.23 %	41.56 %
Average	22.64 %	32.13 %	40.03 %	5.20 %
Maximum	24.82 %	35.42 %	41.49 %	12.81 %
Minimum	17.76 %	29.84 %	36.64 %	3.04 %
StDev	2.44 %	1.83 %	1.40 %	3.02 %
Avg/Max	0.91	0.91	0.96	0.41

Picture7: State-profile configuration of nqueens-omp-11-8-8-boada-5 to 35%-65%.

As we can see the parallelization is correct, so we are going to analyze the scalability of our parallelization using the 'submit-strong-omp.sh' script. To analyse the scalability better, now we have a larger size 'n = 13'. And we did not change the script because our better cutoff is 8 which is the default value of it.

```
par4207@boada-1:~/lab4$ qsub -l cuda submit-strong-omp.sh
nqueens
```

After submitting it, we obtain these plots:



Plot1: Average elapsed execution time and speed-up.

As we can see, the execution time is reduced when we increase the number of threads.
Then, in the speed-up plot, it is seen that despite of not increasing a lot when we increase the number of threads, it is a really increase.

2.1 Optional

In our case, we did not have to change many things, because our worksharing constructs were already inside the task, so we only changed the site of the creations of the tasks, over the 'for' loop.

```
void nqueens(int n, int j, char *a, int depth) {
     int i;
     if (n == j) {
     /* put good solution in heap. */
          if( sol == NULL ) {
                sol = malloc(n * sizeof(char));
                memcpy(sol, a, n * sizeof(char));
          }
          #pragma omp atomic
          sol_count += 1;
     } else {
          /* try each possible position for queen <j> */
          #pragma omp for
          for ( i=0 ; i < n ; i++ ) {
            char *b = alloca (n * sizeof(char));
            memcpy(b, a, n * sizeof(char));
            b[j] = (char) i;
            if (ok(j + 1, b)) {
                nqueens(n, j + 1, b, ++depth);
            }
          }
     }
     #pragma omp taskwait
}
```

Code6: Optionl version of nqueens-omp.

Conclusions

We have observed that our code have a great overhead which does not allow it to have a better performance, as we can see in the speed-up plot of scalability as well as, in the trace-profile that we have obtained in paraver.
The problem of the overhead that we have it is because of we have the 'alloca' and 'memcpy' inside the parallel part, so the program will work better if we put these calls outside of the task.

In the following code there is our first version of the 'nqueens-omp' which we think can obtain a great performance. But due to some compilation errors we had to make another less efficient version.

```
void nqueens(int n, int j, char *a, int depth) {
     int i;
     if (n == j) {
      #pragma omp critical
      {
      /* put good solution in heap. */
          if( sol == NULL ) {
            sol = malloc(n * sizeof(char));
            memcpy(sol, a, n * sizeof(char));
          }
          sol_count += 1
      }
     } else {
          /* try each possible position for queen <j> */
          for ( i=0 ; i < n ; i++ ) {
              char *b = alloca(n * sizeof(char));
              memcpy(b, a, n * sizeof(char));
              #pragma omp task fianl(depth >= CUTOFF) mergeable
              {
                a[j] = (char) i;
                if (ok(j + 1, a)) {
                   nqueens(n, j + 1, b, depth++);
                }
              }
          }
          #pragma omp taskwait
     }
}
```

Code7: First version of nqueens-omp.