PAR Laboratory Assignment

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

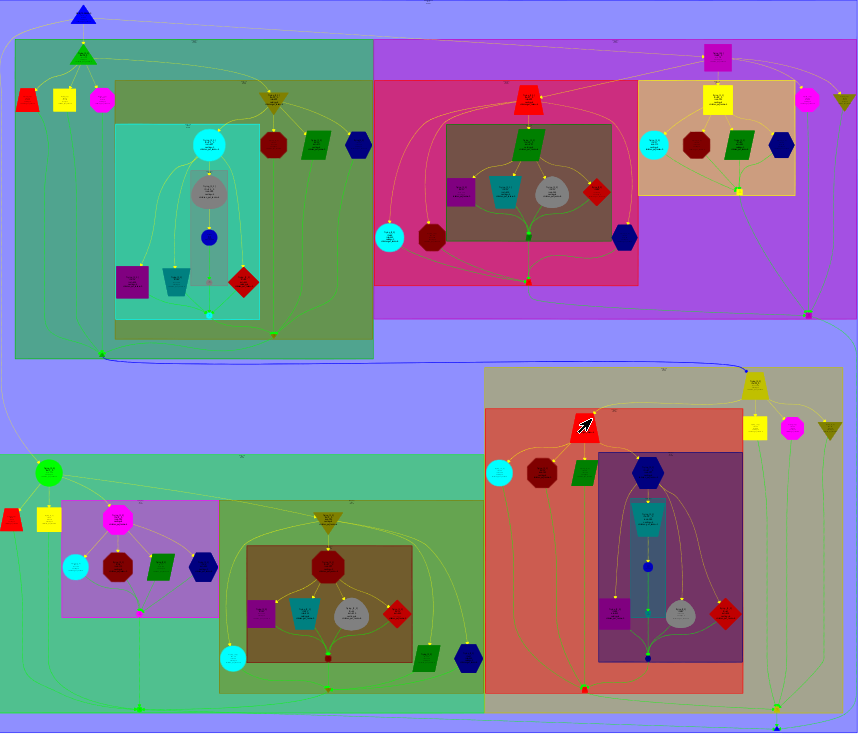
Sandra Flores Hidalgo

Celia Fernández Pérez

par1215

Index

1. Understanding the potential parallelism in Nqueens
2. Shared-memory parallelization
3. **Understanding the potential parallelism in Nqueens:**

****

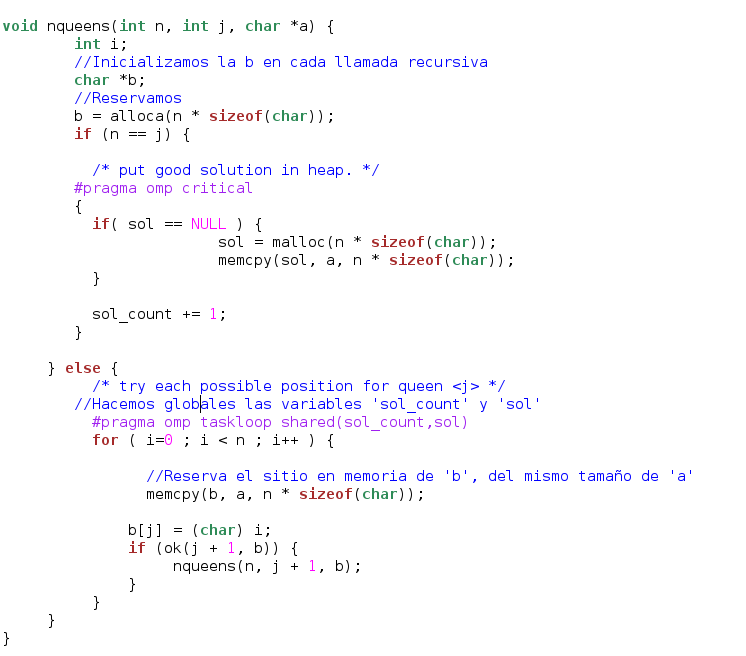
We find the solution task, it is in top-left section and in bottom-right section. The program with 4 queens only have 2 solution then, the graphic only have 2 solution tasks.

In top-left section find the first solution, it is because this section have dependence with the bottom-right section. Because the two section have to shared the “sol\_count” variable.

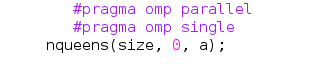
The program try to place the first queen, in all of the places of the board, when it do it, the program try to place the second queen, and the same with the rest of queens, when it arrives at the last queen and do not achieve the program start again and try with other positions. When it achieve to place all of queens it entry in solution task and plus one “sol\_count” and generated a dependence with the next try to place the queens.

The program try to place the queens as may times as the board rows have.

**2. Shared-memory parallelization:**

****

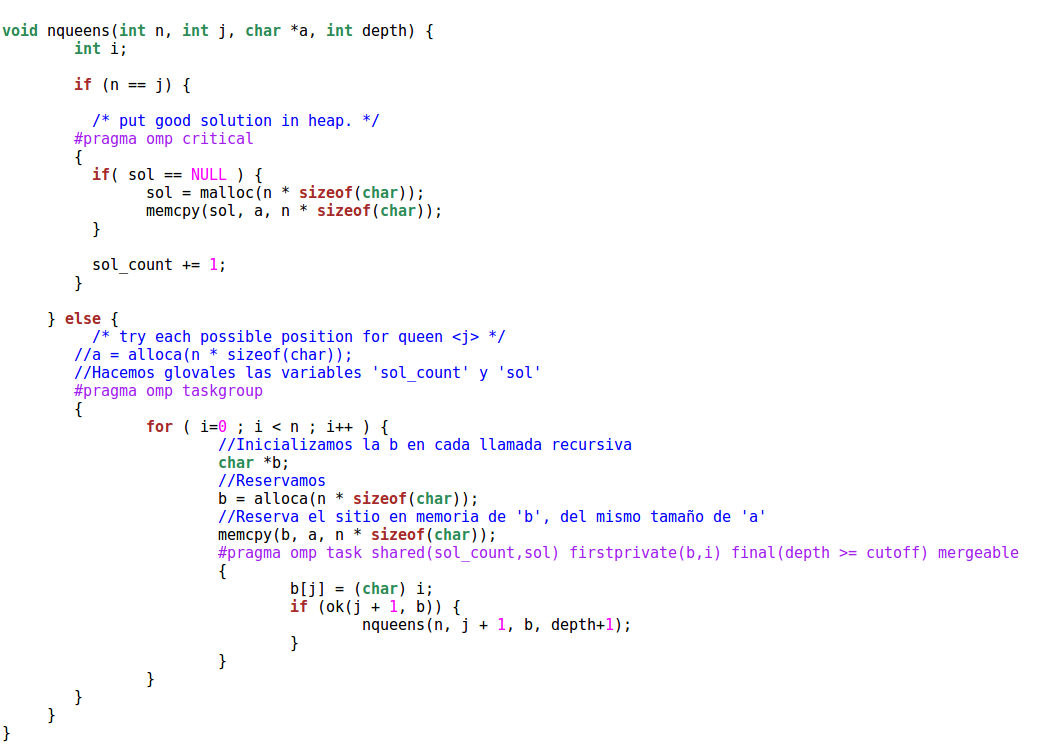
In main:

****

Firstly we add #pragma omp critical to avoid that differents tasks could access to the sol or sol\_count variables at the same time and modify them. Later we use the #pragma omp taskloop shared(sol\_count, sol) to create a task for each loop and the shared clause allows us to define both sol\_count and sol as global variables, so all of the tasks can access to the same value.

In this loop we work with the b variable, it is define at the beginning of the function as “char \*b;”. Later, using “alloca”, we reserve the memory space and then, in the loop we copy the content of a into b, with the function “memcpy”.

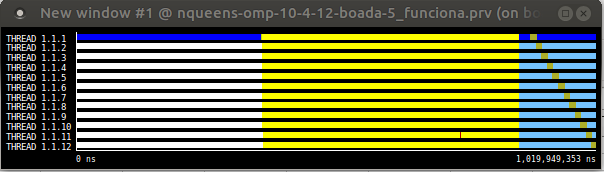
**Code with cut-off:**

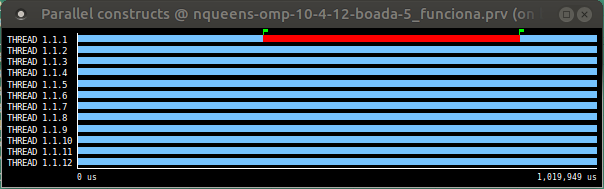
****

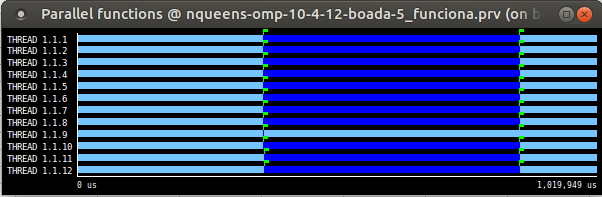
To work with the cutoff version we have to change our previous code because with the taskloop we weren’t able to work with the cutoff clause. So we decided to swap the taskloop for a task generation on each one of the loops. This allows us to guarantee that, when the condition defined final become false, any more tasks are generated.

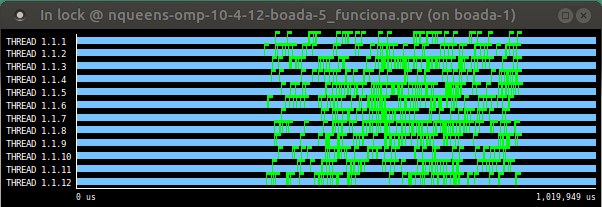
For each task we have to define variables b and i as private, because each one of the tasks have its own value. Also the sol and sol\_count must be shared for every task. We add too the clause mergeable so if the task is undeferred task or included task, amerge task might be generated.

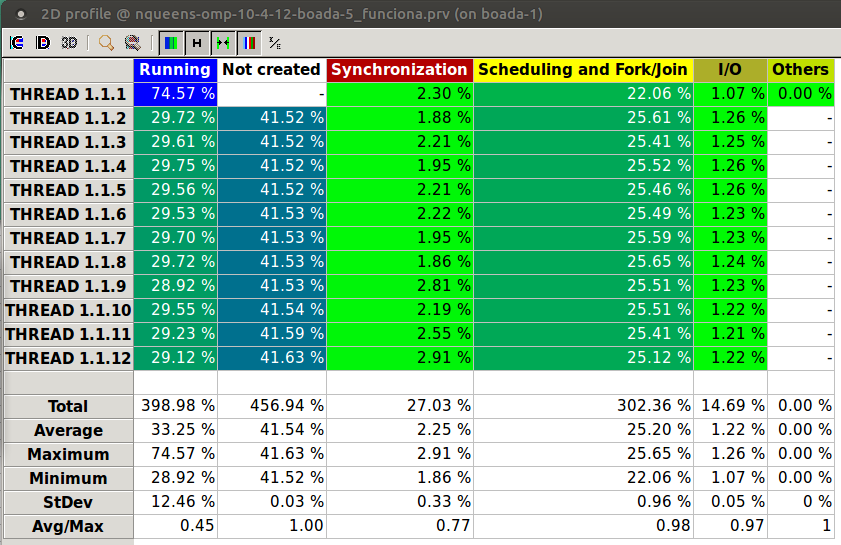
**Paraver graphics:**

****

****

****

****

****

On the first paraver graphic the paraver show us the states of all of the threads. Analyzing it we can appreciate that the master thread, the thread 1.1.1, creates and start running.

When the parallel region arrives, during the for, where the pragma statement is used, all the threads are created and the synchronization region starts, we can see this more precisely on the second paraver graphic. In this we can observe the sync region as the red fragment of the master thread execution. With the third graphic it proves that, while in the sync region the threads are running in parallel. During this region all the threads in one point, with this image we can’t appreciate, goes on a synchronization state too, we can see it on the table.

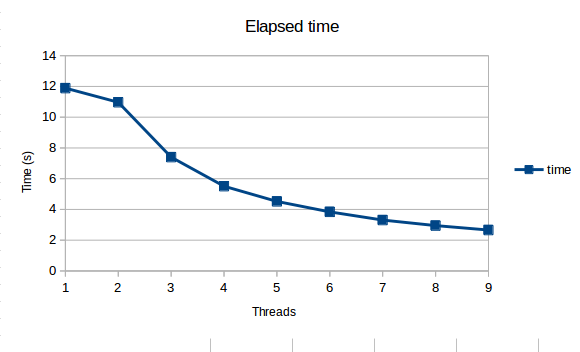
So, after this region all the threads come back to the idle state except the master one that keeps running. After this each one of the threads is going to execute for a moment the I/O Activity.

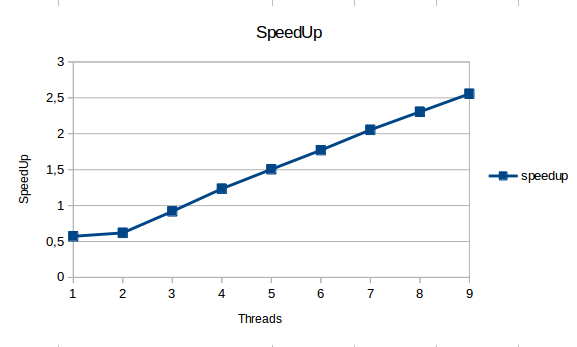
On the fourth image the flags are shown, these allows us to see when the critical region starts and ends.

The percentages of use of each one of the states for all the threads appears on the table. This proves that all of the threads execute a synchronization region in some point, and shows too the I/O activity.

**Speedup and elapsed time:**

The file that has to create the graphics was not generated them, but it generated the text files with the values, for this reason we created them.

****

****

As we can see on the graphics result the parallelization allows us to reduce the time results, showed in the elapsed time one, at the same time that the number of threads grow we can appreciate that the time reduces. Analyzing the speedup it is increasing as the number of threads grows because having more threads in this case implies a higher parallelization. The improvement of this is bigger than the costs from creating the threads, so, we are optimizing the code right.