PAR Laboratory Assignment

Course 2018/19 (Fall semester)

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

Pablo Vizcaino, Guillem Ramírez

User: Par2206

December 3, 2018

Contents

1 Introducion

In this labaratory assignement we will be working with the N-queens problem. This problem states that we have to find all possible solutions for placing N-queens in an $N \times N$ chessboard so that each queen can't attack any of the other queens placed.

The workarround for this problem is to use a Branch and Bound technique. This technique tries all possible combinations with a recursive algorithm using a backtracking aproach.

Take into account that the computational cost of this problem is expensive by reason of it has to compute all combinations in a chess board. For example, in an 8×8 chess board the algorithm tries $\binom{64}{8}$ combinations, aproximately 4.42×10^9.

Consequently, we will be analysing the potential parallelism of the N-queens problem and proposing a parallel version of the code, hopefully improving it's performance.

2 Task decomposition analysis

In this section we are going to take a look into the task decomposition analysis of the nqueens algorithm.

First of all we runned a tareador instrumented version of the code. Which gave the following output.

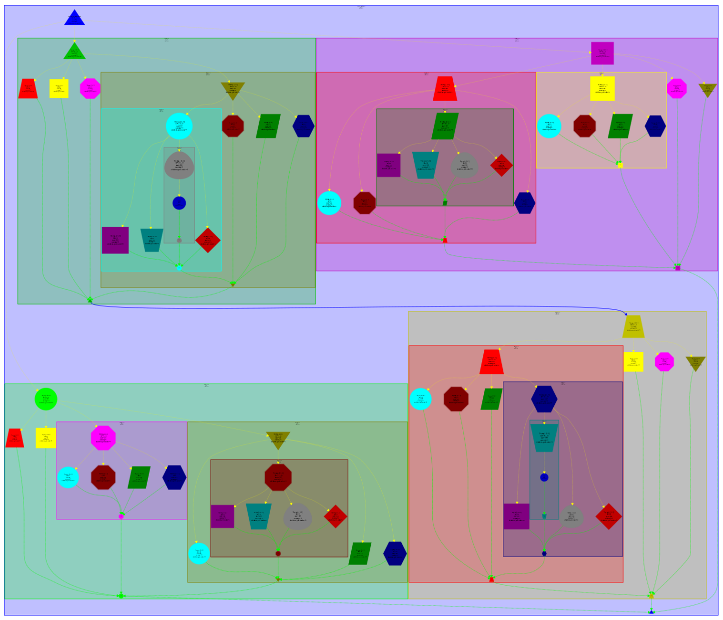

Figure 1: Strong speed-up plots

From the dependency graph we can visually observe the execution of the backtracking algorithm. Also we observe only dependencies between a call to nqueens and the function that calls it, as they share the vector `char *a`. We must take that into account for our future parallel implementations.

Another thing that we observe from the graph is that for each recursion level we create a lot of tasks. We probably will need to implement a cutoff mechanism to reduce overheads due to task creation and work distribution.

3 Parallelization strategy

After evaluating the task decomposition and choosing which region we are going to parallelize, we have to choose a strategy. Since the approach to the n-queens problem given to us is recursive, we have to choose between the two strategies that have been taught to us; leaf or tree.

3.1 Task creation and synchronization

Looking at the code it's clear that the tree strategy will be more suited to the problem, since the base case doesn't do a huge amount of calculations and each recursive call does parallelizable work (a for-loop and memory copy), so creating tasks as we go through the recursion will increase the performance.

In the following code you can see our initial approach (keep in mind that this version isn't finished yet):

```
void nqueens(int n, int j, char *a) {
    int i;
    if (n == j) {
        if( sol == NULL ) {
            sol = malloc(n * sizeof(char));
            memcpy(sol, a, n * sizeof(char));
        }
        sol_count += 1;
    } else {
        for ( i=0 ; i < n ; i++ ) {
            #pragma omp task
            {
                char * new_a = alloca(n * sizeof(char));
                memcpy(new_a, a, j*sizeof(char));
                new_a[j] = (char) i;
                if (ok(j + 1, new_a)) {
                    nqueens(n, j + 1, new_a);
                }
            }
        }
        #pragma omp taskwait
    }
}
```

We decided to create one task for each iteration of the for-loop and then, to satisfy dependences, synchronize them at the end of the loop with a taskwait. One important change from the initial version is the use of `alloca` and `memcpy`. Each task is reading and writing the memory to check if the possible position of the queen is correct, and each of them has to do so independenly from other tasks, so what we have chosen to do to solve the problem is to copy the vector `a` for each task. This will give us some overhead, as copying memory is not a fast operation.

3.2 Race conditions

Having finished the task creation and synchronization of the code, there is still work to do. The last version of the code doesn't give the correct result yet due to two race conditions. The first one and the easier to solve is the increment of the solution counter variable. This global variable is read and then written by each task, and since two of them can access it at the same time (giving a different outcome depending on the order) we are in front of a race condition problem.

To solve it, we placed a `#pragma omp atomic` before the operation so we guarantee that only one thread is doing the increment at a time.

```
    }
    #pragma omp atomic
    sol_count += 1;
} else {
```

With this change, the execution of the code gives the correct answer to the problem regarding the total number of possible solutions. However, it doesn't print the same solution each time and looking at the code, the solution may even we wrong.

This is due to the second race condition. Two or more threads can enter the `if (sol==NULL)` part of the base case. Not only will this threads be the same each time (since it depends on how fast they traveled the recursive tree) but they can be copying the memory a the same time, giving a mixed solution (some parts of the solution of one thread and some parts of another one).

The initial fix to this problem is to declare a critical region that protects the whole if statement. This would work fine, but it has some overhead. each thread that gets to the base case will have to synchronize and wait to check the condition, even though we know that once a solution is found it will never be false again. We can optimize it doing a double-check before entering the critical zone :

```
if( sol == NULL ){
    #pragma omp critical
    if( sol == NULL ){
        sol = malloc(n * sizeof(char));
        memcpy(sol, a, n * sizeof(char));
    }
}
```

The critical zone mentioned before is now inside another if statement; this one not protected by a critical zone. Doing this only the firsts threads to reach the base case will enter to the critical zone. The other threads will not enter inside the first if statement so they wont execute the critical pragma.

The final state of the code, in this section, is the following:

```
void nqueens(int n, int j, char *a) {
    int i;
    if (n == j) {
        if( sol == NULL ) {
            #pragma omp critical
            if( sol == NULL ){
                sol = malloc(n * sizeof(char));
                memcpy(sol, a, n * sizeof(char));
            }
        }
        #pragma omp atomic
        sol_count += 1;
    } else {
        for ( i=0 ; i < n ; i++ ) {
            #pragma omp task
            {
                char * new_a = alloca(n * sizeof(char));
                memcpy(new_a, a, j*sizeof(char));
                new_a[j] = (char) i;
                if (ok(j + 1, new_a)) {
                    nqueens(n, j + 1, new_a);
                }
            }
        }
        #pragma omp taskwait
    }
}
```

3.3 Worksharing constructs

Finally we tried to implement a version of the code using worksharing constructs. Getting the code working is not an easy job.Since it's within a recursive function, we have to take into a account that parallel regions cannot be nested so we need to declare one `#pragma omp parallel` before each construct.

First of all we implemented an initial version using a `taskloop` construct as a middle step for our objective.

```
                /* try each possible position for queen <j> */
                #pragma omp parallel
                #pragma omp taskloop final(j ¿= cutoff) mergeable
                for ( i=0 ; i < n ; i++ ) {
                        char * new_a = alloca(n * sizeof(char));
                        memcpy(new_a, a, j*sizeof(char));
                        new_a[j] = (char) i;
                        if (ok(j + 1, new_a)) {
                                nqueens(n, j + 1, new_a);
                        }
                }
        }
}
```

The transition from the `taskloop` version to the worksharing construct was not easy.

As the for OpenMP construct has no final or if clauses, for implementing the cutoff mechanism we must change the schedule of the for in runtime. For doing so, we included a if mechanism that in runtime checks if the depth exceeds the cutoff and then runs an static version of the for or an OpenMP instrumented one.

The version using worksharing constructs looks like:

```
void nqueens(int n, int j, char *a) {
        int i;
        if (n == j) {
                /* put good solution in heap. */
                if( sol == NULL ) {
                        #pragma omp critical
                        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> */
        if(j>=cutoff){
            for ( i=0 ; i < n ; i++ ) {
                char * new_a = alloca(n * sizeof(char));
                memcpy(new_a, a, j*sizeof(char));
                new_a[j] = (char) i;
                if (ok(j + 1, new_a)) {
                    nqueens(n, j + 1, new_a);
                }
            }
        }else{
            #pragma omp parallel
            #pragma omp for
            for ( i=0 ; i < n ; i++ ) {
                char * new_a = alloca(n * sizeof(char));
                memcpy(new_a, a, j*sizeof(char));
                new_a[j] = (char) i;
                if (ok(j + 1, new_a)) {
                    nqueens(n, j + 1, new_a);
                }
            }
        }
    }
}
```

4 Performance evaluation

In this section we are going to evaluate our parallel approach, generate and comment some graphs and also implement a cut-off mechanism.

The execution time of the last version is 1.065s. If we compare it to the execution time of the serial version, 0.942s, we can see that the performance got a 13% worse. This is caused by the serious overheads of task creation and synchronization plus the memory copy of each thread. In the following figure you can see the time diagram from paraver with a size of 4 and 8 threads.

In order to get a better performance and as we seen on previous laboratories, we are going to implement a cut-off mechanism.

We only have on pragma creating tasks, so the cut-off will be fairly easy to implement. To control how deep we are in the recursion tree we don't need to add an extra variable, since the *j* parameter already gives us that information. We used this value to implement a final clause in our task creation pragma, so once it reaches a certain level it stops creating more tasks.

```
#pragma omp task final(j >= cutoff) mergeable
```

In order to obtain the optimal number for the cut-off value, we created a script that tried different values for a given number of threads.

```
setenv OMP_NUM_THREADS 8
setenv OMP_WAIT_POLICY "passive"
setenv size 12
set depth_list = "0 1 2 3 4 5 6 7 8 10 15 20 32 64 128 1023"
set out = out.txt
rm -rf $out
rm -rf o
rm -rf ./elapsed.txt
foreach depth ( $depth_list )
    echo $depth >> $out
    ./nqueens-omp -n$size -c$depth >> o
    set result = `cat o | tail -n 4  | grep "Solution Count Time"
                 | cut -d' ' -f 5`
    echo $result >> $out
end
```

We ran this script changing the number of threads to obtain the following table:

	Cut-off value								
ThreadsThreads	1	2	4	6	8	16	32	64	512
4	0.73	0.72	0.71	0.81	1.33	2.03	2.03	2.04	2.0
8	0.45	0.44	0.42	0.49	0.71	1.12	1.08	1.11	1.09
12	0.38	0.34	0.36	0.41	0.51	0.83	0.81	0.85	0.86
16	0.36	0.29	0.25	0.29	0.44	0.68	0.68	0.68	0.67

We can observe that the optimum value of the cut-off variable is 4 in most cases. With this value and the same number of threads(8) than when we evaluated the omp version without cut-off, we obtain an execution time of 0.42s. This marks a speed up of 2.53 to the first parallel version and 2.24 to the serial version.

To finish evaluating the parallelization of the code, we are going to obtain the strong scallability plot from the last version; using a size of 13, 8 threads and a cut-off value 4.

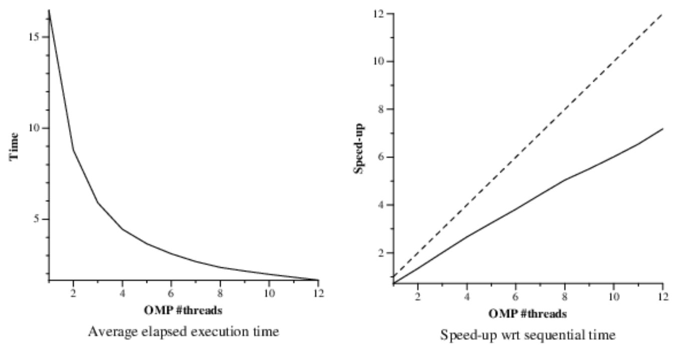

Figure 2: Strong speed-up plots

As it can be seen, the strong scallability is not as close to the ideal case as we are used to. This is caused by the overhead of the memory copying, the critical/atomic zones and the cut-off mechanism.

5 Conclusions

In this assignment we learnt how to analyze a more complex algorithm and parallelize it, with an extra problem that we hadn't found on the previous labs; the problem with memory copying and the usage of *alloca* function. We also learnt how to re-use the cut-off script from the previous lab and adapt it to the current problem.

Another issue that we could observe very clearly in this assignment is the importance of a good cut-off mechanism, as the execution time of the parallel version increased the time compared to the serial one but greatly improved when using an optimum cut-off value.