Comparison on NQueens problem parallelization using OpenMP and OpenCL

Juan Alcantara Dominguez (student number: 40488349)
Edinburgh Napier University

Abstract

This report is a description on implementation and results of the parallelization of NQueens problem using CPU and GPU parallelization. Both implementations were compared between each other as well as with a non-parallelized version of the code. Results are discussed and improvements for further experimentation are proposed.

Contents

Introduction

The NQueens problem consists of placing N queens in an N*N chessboard in such a way that none of the queens is attacked by another queen. This problem has been around for a while and many implementations of parallel solutions have been studied and suggested. The main reason for this is that, for implementations after N > 19, processing time would rises to over an hour, as it is the case in a study by Thouti and Sathe (2012), on the parallelization of nqueens problem with OpenCL. The challenge today persists to create a more efficient and faster Nqueens solution using parallel programming.

Implementation

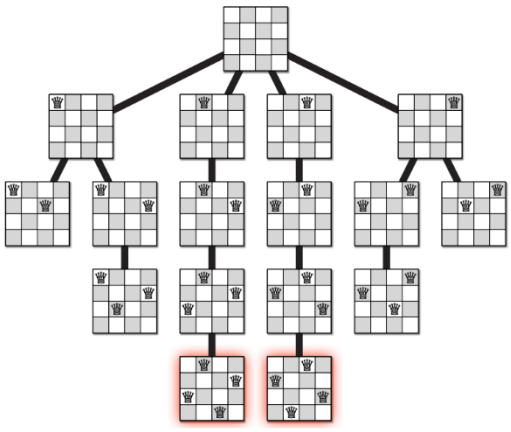
This section will go through the implementation of the code with OpenMP and OpenCL. For this experiment the latest stable versions of OpenMP and OpenCL were used. Two main challenges were the task separation for threads and the elimination of recursive methods for the GPU parallelization.

OpenMP

For CPU parallelization with OpenMP, an *Intel(R) Core(TM) i5-8350U CPU @ 1.70GHz 1.90 GHz* and *8GB RAM were used.*

First Implementation

The first implementation of OpenMP parallelization was done with the provided sample code. The parallelization on this code was done by sending to each thread the gameboard with the first queen placed. The maximum number of threads would be N in an N*N chessboard.



[Fig 1] Backtracking explored branches.

Figure 1 clearly shows the branches that are explored with the backtracking algorithm. The parallelization of the sample code would send each child node of the root node to a thread in the CPU, so that each subtree was explored by each thread.

```
80  void calculateAllSolutions(int N, bool print)
81  {
82      std::vector<std::vector<int>> solutions;
83      std::vector<int> gameBoard(N, 0);
84      #pragma omp parallel for
85      for (int i=0 ; i<N ; i++)
86      {
87          calculateSolutionsBridge(i, gameBoard, N, solutions);
88      }
89
90      printf("N=%d, solutions=%d\n", N, int(solutions.size()));
91
```

[Fig 2] OpenMP parallelized loop for first implementation.

Figure two shows the most important line in the first implementation. The #pragma before the loop will parallelize the cycles of the loop. This is done automatically using omp.h library. Function called in each thread is calculateSolutionsBridge().

```
73  void calculateSolutionsBridge(int firstRowCol, vector<int> gameBoard, int N, std::vector<std::vector<int>>& solutions){
74      gameBoard[0] = firstRowCol;
75      //Initiate recursive on second row function inside thread with allocated first row queen
76      calculateSolutionsThread(1, gameBoard, N, solutions);
77  }
```

[Fig 3] Bridge function first implementation.

The bridge function is the first code runed by the thread. It will add the first queen to the first row and call the recursive function starting on the second row. It can be seen in figure 3, how the gameboard is not passed to the bridge function as a reference. This way each thread will have a unique variable.

```
52  // A recursive function to calculate solutions
53  void calculateSolutionsThread(int writeToRow, std::vector<int>& gameBoard, int N, std::vector<std::vector<int>>& solutions)
54  {
55      // for each column
56      for (int i = 0; i < N; ++i)
57      {
58          // set queen at the current column, at the specified row (writeToRow variable)
59          gameBoard[writeToRow] = i;
60          // if the board is valid so far
61          if (boardIsValidSoFar(writeToRow, gameBoard))
62          {
63              const auto nextWriteToRow = writeToRow + 1;
64              if (nextWriteToRow != N) // haven't filled the chessboard yet, so fill the next row
65                  calculateSolutionsThread(nextWriteToRow, gameBoard, N, solutions);
66              else // filled the chessboard, so save the solution
67                  #pragma omp critical
68                  solutions.push_back(gameBoard);
69          }
70      }
71  }
```

[Fig 4] Recursive function for first implementation.

Figure 4 shows the recursive function that runs inside each thread. It can be seen how gameboard is now referenced to the gameboard in the bridge function. The other important aspect of the code is in line 67, before adding the solution. Since all threads add to the same solution vector this must be protected.

Second Implementation

Another implementation was made without recording the solutions and instead just counting the number of solutions. The fact that the solutions didn't have to be recorded, allowed the program to be a lot more efficient by saving memory usage and processing time.

```
47  uint64_t solve( int N )
48  {
49   uint64_t solutions=0;
50   int row=N-1;
51
52   #pragma omp parallel for reduction(+:solutions)
53   for (int i=0;i<N;i++)
54    solutions+=solveParallel(row-1,N, (1ull << (32+i-row)), (1ull << (row+i)), (1ull << i ));
55
56   return solutions;
57  }
```

[Fig 5] Second Implementation central method.

Like the first implementation, multithreading was done by sending to each thread each of the first queen positions. However instead of writing and locking solutions variable this method will return and integer containing the number of solutions. "reduction()" clause will ensure each thread gets its own copy of solutions variable and they are all added correctly.

```
29  uint64_t solveParallel( int row, int N, uint64_t diag45, uint64_t diag135, uint64_t cols)
30  {
31   uint64_t solutions=0;
32   if (row)
33   {
34    for (int i=0; i<N; i++)
35     if (test(row,i, diag45, diag135, cols))
36       solutions+=solveParallel ( row-1, N,
37                          diag45 | (1ull << (32+i-row)),
38                          diag135 | (1ull << (row+i)),
39                          cols | (1ull << i ));
40   }
41   else
42   for (int i=0; i<N; i++)
43     solutions += test(0,i, diag45, diag135, cols);
44
45   return solutions;
46  }
```

[Fig 6] Recursive function for second implementation.

Recursive function for second implementation receives several attributes. Current row, N, occupied 45-degree diagonals (diag45), occupied 135-degree diagonals and occupied cols. It uses unsigned 64-bit integers and bitwise operations to specify the unavailable columns and diagonals. This makes the program faster and more efficient memory wise but also makes it very dependent on recursion.

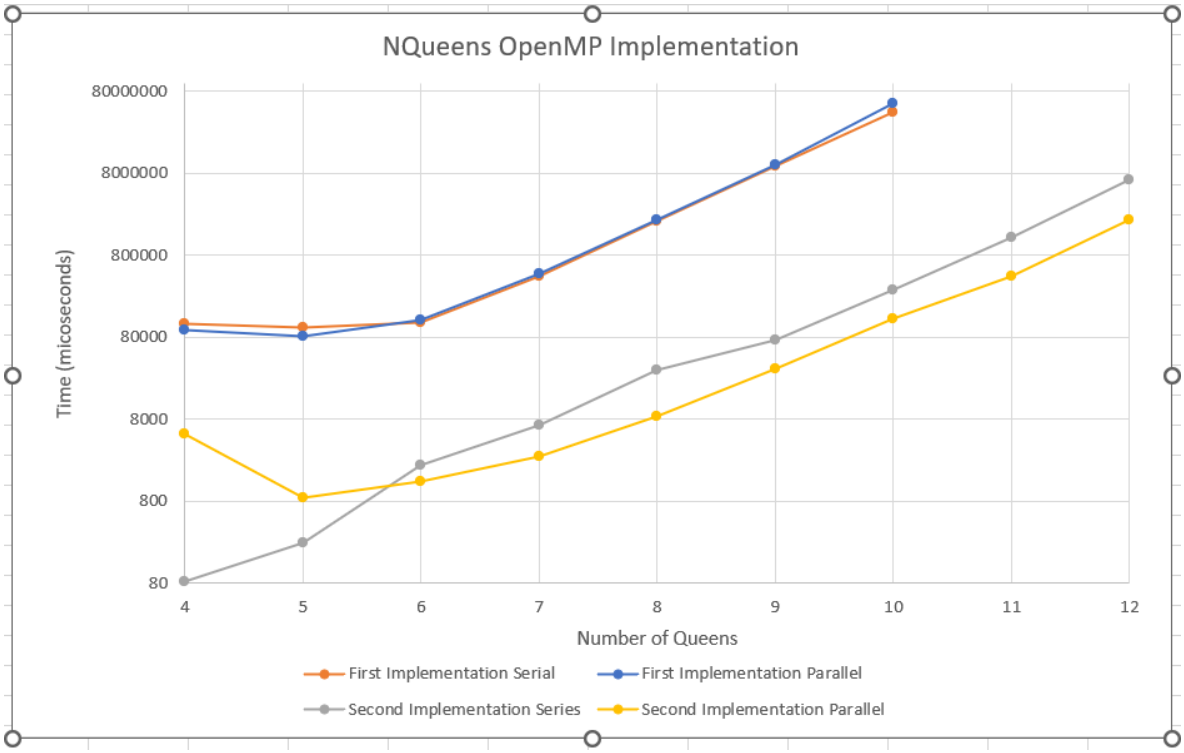
Finally, Test() function check that the column position in the next row is available, and continues recursion if true.

Results

For OpenMP performance recordings, code was runed 50 times for each test. This results in higher reliability at the point of comparing different implementations.

First and second; serial and parallel

We will first compare the serial and parallel versions of the first implementations with the serial and parallel versions of the second.



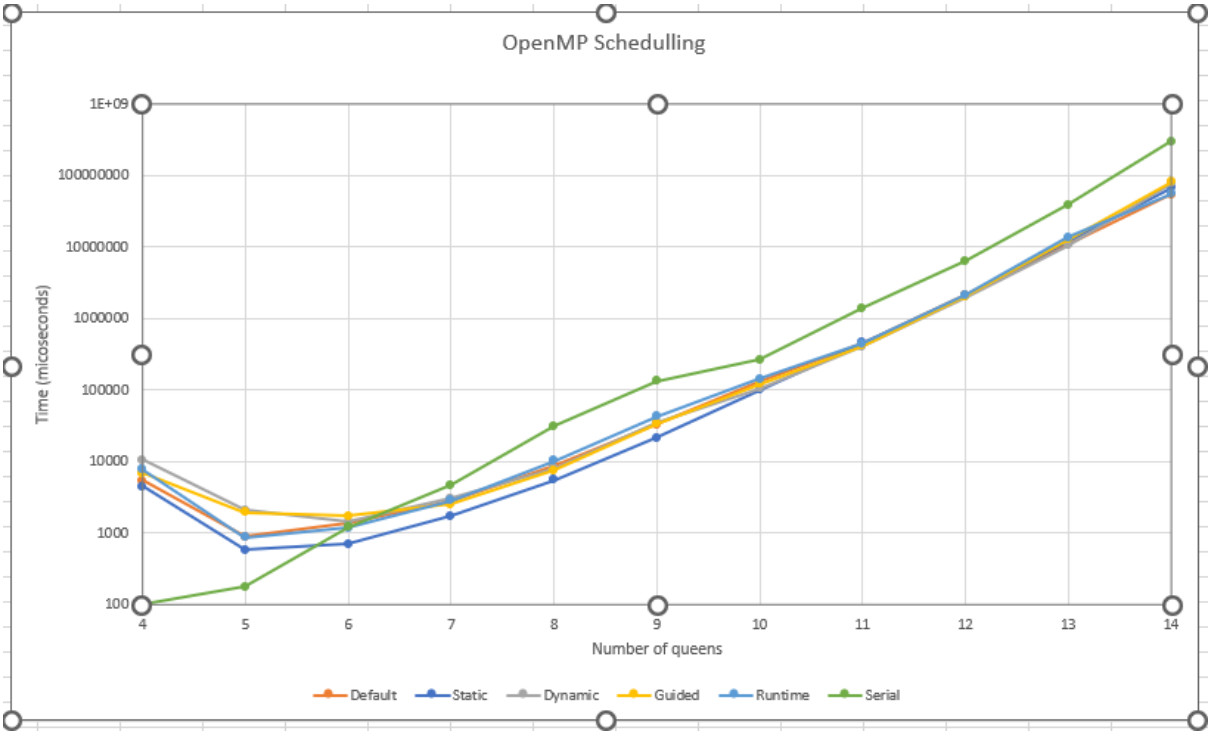
[Fig 7] Logarithmic table of first results

Little improvement can be seen with the first implementation. Actually, the graph implies that with n-queens >10 the serial version could be faster (higher numbers of queens where not tested due to excessive processing time), however further testing would be needed in order to say this with confidence.

For the Second implementation however, it can be observed how a constant performance increase is achieved for n-queens > 6. For example, for n-queens = 12 there is a 3 times speed increase achieved with the parallel version of the code, from 6690000(3s.f.) microseconds to 2160000(3s.f.) microseconds for 50 iterations of the code.

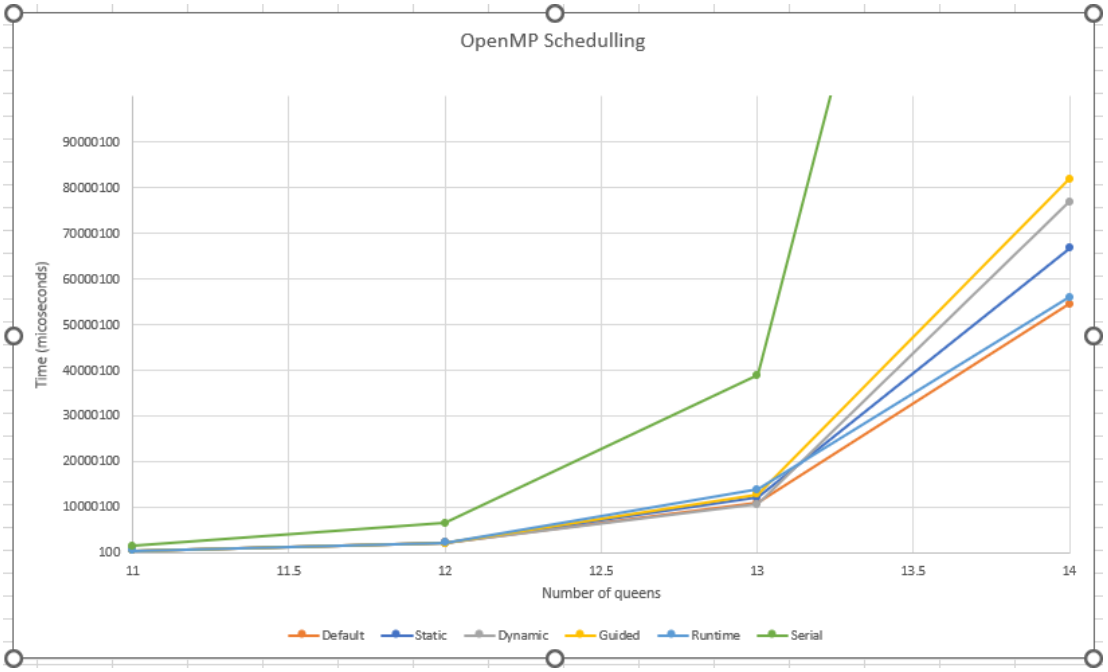
Scheduling methods for second implementation

Since observations with second implementation where significantly better, a deeper analysis was made on different scheduling methods.



[Fig 8] Logarithmic graph different scheduling types

After running code 50 times for each n-queen value and each scheduling type little difference can be seen in figure 8.



[Fig 9] Linear view n-queens 11 to 14

However, a more precise look at n-queens 11 to 14 with a linear time scale, shows a significant performance advantage for default and runtime scheduling types of OpenMP, and a significant disadvantage for guided and dynamic. Exact times for 50 iterations with 14 queens are: Guided - 82s, Dynamic – 77s, Static – 67s, Runtime – 56s, Default – 55s.

The time with the serial version of the code for 14 queens was 302s. This shows significant improvement was achieved by using OpenMP to parallelize Nqueens problems. It also shows Runtime and Default OpenMP scheduling types to be the most suitable for this version of the code. However, further experimentations and testing should be done in order to better understand the functioning and the processing benefits of OpenMP with Nqueens problem.

OpenCL

The second part of the experiment focuses on parallelization in the GPU. There are many ways to parallelize the GPU. The options considered for this study were using CUDA and OpenCL, and eventually, OpenCL was chosen due to the complicated access to an NVIDIA GPU.

Implementation

Implementation of parallelization with OpenCL wasn't as simple as OpenMP and caused many time-consuming problems during development. This is due to the complexity of data management when programming on the GPU and the difficulty debugging the kernel code. The final prototype will calculate the number of solutions without saving the individual solutions to a list. However, unlike the second OpenMP implementation, this program does allow for solutions to be exported since it backtracks using an array symbolizing the N*N board.

Host Code

```
try
{
    // Get the platforms
    vector<Platform> platforms;
    Platform::get(&platforms);


    // Assume only one platform.  Get GPU devices.
    vector<Device> devices;
    platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);

    // Just to test, print out device 0 name
    cout << devices[0].getInfo<CL_DEVICE_NAME>() << endl;

    int WorkGroupSize = devices[0].getInfo<CL_DEVICE_MAX_WORK_GROUP_SIZE>();

    // Create a context with these devices
    Context context(devices);

    // Create a command queue for device 0
    CommandQueue queue(context, devices[0]);
```

[Fig 10] Retrieving device data. OpenCL initiation.

The host code starts by retrieving the device data. It will get the [0] (first) device from the list of devices which should generally be the main GPU if not the only one. For this experiment an *Intel(R) UHD Graphics 620* was used with *Max Memory: 3210MB.*

```
52          // Create the buffers
53          Buffer bufSolutions(context, CL_MEM_WRITE_ONLY, sizeof(int));
54          Buffer bufBoard(context, CL_MEM_READ_WRITE, N * sizeof(int));
55          // Copy data to the GPU
56          queue.enqueueWriteBuffer(bufBoard, CL_TRUE, 0, N*sizeof(int), &gameBoard);
```

[Fig 11] Buffer initiation and enqueuing.

Next, buffers are initiated for the memory to be allocated. Buffers used where the solutions write buffer, where number of solutions would be written by all kernels, and the gameboard buffer, allocated in each kernel instance for the calculation of solutions.

```
93          */
94          queue.enqueueNDRangeKernel(kernel, NullRange, global, local);
95          // Copy result back.
96          queue.enqueueReadBuffer(bufSolutions, CL_TRUE, 0, sizeof(int), &solutions);
97          cout << "N: " << N << " solutions: " << solutions << endl;
98
```

[Fig 12] Creation of kernel and retrieving of results.

Finally, the buffer is created, and solution is read. Different global and local sizes were tested for best results. Finally global being equal to N of queens and local being 1 would generate the best results. Kernel code should be adapted accordingly for relevant changes.

Kernel Code

```
66  __kernel void solve(const int N, __global int *solutions, __local int *gameBoard)
67  {
68      int idx = get_global_id(0);
69
70      if (idx >= N) {
71          return;
72      }
73
74      ulong sols = 0;
75
76      for (int i = 0; i < N; i++) {
77          gameBoard[i] = -1;
78      }
79      gameBoard[0] = idx;
80
81      sols = solveNonRecursive(gameBoard, N);
82
83      atomic_add(solutions, sols);
84  }
```

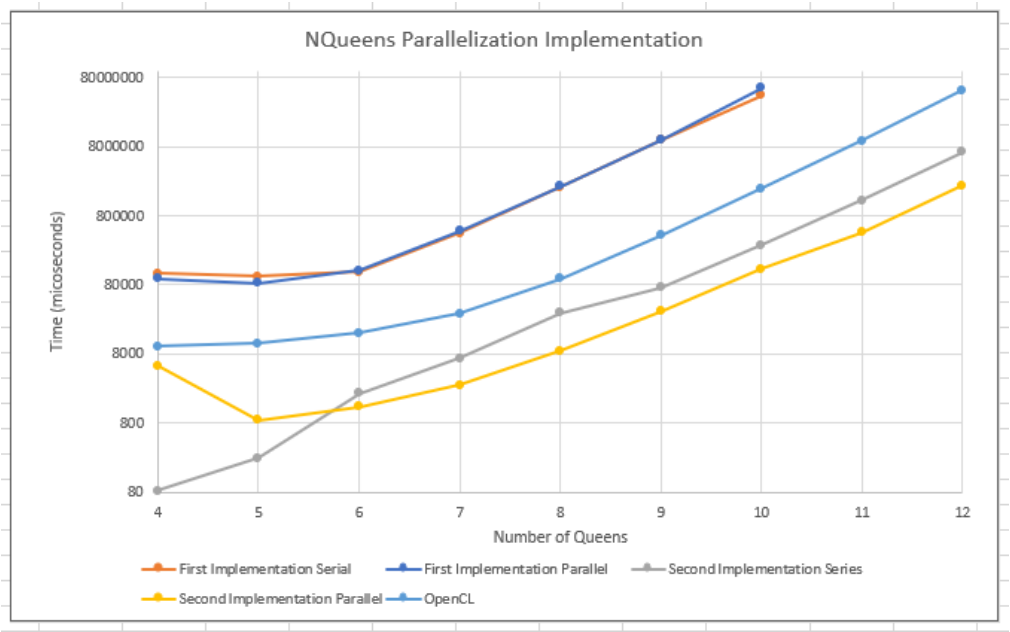
[Fig 13] Kernel method for each subtree

Main kernel method with retrieve the global id of the thread, initiate the gameboard and place the respective queen on the first row, being the queen the global id of the thread. It will call the non-recursive solver for the rest off the rows and carry out an atomic_add to the solutions, to protect the integrity of the variable solutions.

The figure of the non-recursive code can be seen after the references.

Results

A comparison between the performance of OpenCL implementation and OpenMP.



[Fig 14] OpenCL Implementation compared with others

When OpenCL data is added to the graph with the other implementations it doesn't seem to stand out considering the added difficulty of implementation. However, taking into consideration the added functionality that it has compared to the second OpenMP implementation, these results can be considered positive. OpenCL implementation has the potential of the first implementation allowing all solutions to be saved with a significant increase in performance. Time for 10 queens drops from 56 seconds in first OpenMP implementation to 2 seconds using OpenCL.

Evaluation & Conclusions

Both for OpenMP and OpenCL implementations, the first N locations of the queen in the first row were parallelized. This doesn't fully utilize the parallelization potential that OpenMP and OpenCL offer, despite producing a significant performance boost. The main reason for this is that different branch sizes result in threads finishing while others continue to process code. A more organized solution would guarantee that completed threads have a task to execute and therefore reduce total processing time.

However, the results achieved were significant. If the goal was merely to calculate the number of solutions, an implementation like the second with OpenMP would outperform a simpler serial version or even a GPU parallelization that also calculated solutions themselves. However, when calculating the actual solutions, GPU programming resulted to be the most efficient with a 20x speed increase compared to the equivalent OpenMP version.

Many of these results may also be affected by the differences in the code and the algorithms used since these had to be adapted to suit each implementation. Hence further

research would be needed to precisely discover the potential of OpenMP and OpenCL, but this experiment shows that the Nqueens problem can benefit greatly from parallel programming in both the CPU and the GPU.

References

K. Thouti and S. R. Sathe, "Solving N-Queens problem on GPU architecture using OpenCL with special reference to synchronization issues," 2012 2nd IEEE International Conference on Parallel, Distributed and Grid Computing, Solan, India

```
int solveNonRecursive(__local int* board, int N) {
    int sols = 0;
    int i = 1;
    int j = 0;

    // Start the loop from the second queen
    while (i < N) {
        bool found = false;
        // Try to place the i-th queen in the next column
        for (j = board[i] + 1; j < N; ++j) {
            board[i] = j;
            if (isStillValid(i, board)) {
                found = true;
                break;
            }
        }

        if (found) {
            if (i == N - 1) {
                // A solution is found
                sols++;
                // Reset the position of the last queen and backtrack
                board[i] = -1;
                i--;
            }
            else {
                // Move to the next queen
                i++;
            }
        }
        else {
            // No valid position found for the i-th queen, backtrack
            board[i] = -1;
            i--;
        }

        // If we have backtracked past the second queen, all solutions are foundfor branch
        if (i < 1) {
            break;
        }
    }

    return sols;
}
```

[Fig 15] Non recursive version of solution calc.