COMP90025 Project 2: N-Queens Puzzle

Zhaoqi Fang (1035276) — zhaoqif@student.unimelb.edu.au
Saier Ding (1011802) — saierd@student.unimelb.edu.au

Introduction

MPI (Message Passing Interface) is an application programming interface that is designed to pass messages among processes in order to perform a task. It works out quite well in practice for parallel applications. OpenMP is an application programming interface which is designed to process massively parallel data on multiprocessor shared memory machines.

The N Queens problem is to place n queens on an N*N chessboard and is a CSP (Constraint Satisfaction Problem) that there are no two queens can be on the same horizontal, vertical or diagonal line. Traditional approaches to solve this problem are based on backtracking method and full permutation algorithm which all two methods can be seen as a form of intelligent depth-first search. While because of the complexity of these methods typically rise exponentially with problem size, these methods only could solve N Queens Problem for n up to 100 (The number of solutions is shown in the following table), so we choose a relatively small data size (n=10) as test case for performance analysis. In this report, we will use MPI and OpenMP to parallel full permutation algorithm in N Queens Problem.

N	1	2	3	4	5	6	7
Solutions	1	0	0	2	10	4	40
Unique Solutions	1	0	0	1	2	1	6
N	8	9	10	11	12	13	14
Solutions	92	352	724	2680	14200	73712	365596
Unique Solutions	12	46	92	341	1787	9233	45752

Table 1. The solutions of N Queens Problem in the different number

Hypothesis and Method

The classic sequential algorithm of the N Queens problem is to use the backtracking method and full permutation algorithm. Based on the different time complexity of these two methods, full permutation has a low time complexity(O(n!)), so we choose the full permutation algorithm as the sequential method to solve this problem. The basic idea of the full permutation algorithm is to place the queen in n positions on the first line, and another queen in n-1 positions in the second line, and so on, so we have n! situations need to be considered, and then through the check function to determine whether there are two queens on the same diagonal line, finally we can get the final number of solutions.

The first hypothesis is using MPI to parallel the sequential code, because this sequential code has many processes that using for loop to calculate the results. In order to parallelize the sequential code, one possible method is to extract the for loop in permute() and set a new for loop in main function which aims to iterate the first queen on diagonal of the board for finding the permutations (e.g the permutations of first queen on A1). Because each diagonal can only have one queen, the solutions of each iteration would not affect others because the check if board is valid method will ignore it. Therefore, it can be used by MPI to distribute each iteration to get the permutations and find possible solutions. The concrete methods are listed below.

1. Get the size of processes and rank of current process.
2. Set for loop index i = rank; i < n; i += size (n is the number of queens).
3. Permute board and store the solutions count for each rank.
4. Use MPI_Reduce to sum the solutions count for each rank to get the total solutions

The second hypothesis is to use OpenMP to parallel the sequential code because OpenMP is based on thread approach, where threads can share memory which could be an advantage compare to MPI. The parallel method using by OpenMP is quite similar to the MPI one which distribute the first queen on diagonal of the board for finding permutations to different threads. The concrete method is listed below.

1. Initiate a new 2D integer board array for different threads
2. Get the max size of threads and the thread number by using omp_get_max_threads() and omp_get_thread_num().
3. Set for loop index i = omp_get_thread_num(); i < n; i += omp_get_max_threads()
4. Use #pragma omp parallel private (i) reduction(+:total) for parallelizing the for loop

Finally, we make a hypothesis that using OpenMP on the basis of MPI can run the sequential code faster and more

efficiently. So we can combine the OpenMP method with MPI as the third experiment on the main process for loop based on MPI.

Results and Analysis

- **Sequential Algorithm (Full permutation)**

Firstly, for the sequential method which uses full permutation algorithm. We design the first experience in order to set the baseline of performance analysis, and the sequential code performance should be considered. The result is shown below.

Size	Elapsed Time
4	26
8	3941
10	452037

Figure 2. The elapsed time (us) of sequential method based on the different size

From figure 1 we can find that using the same method, as n increases, the elapsed time increases exponentially. At the same time, we can see from the figure that when size equals to 10, it needs a long time to calculate, so we will not continue to do it more experiments in sequential method, the latter experiments are also relatively small size(smaller than 10), which is enough to reflect whether the program is optimized.

- **MPI (Full permutation)**

Secondly, we design the MPI interaction based on the first layer of the for loop for the main function, because the outermost layer of the for loop in the sequential method is to traverse the chessboard of each row, so that each queen is in the different row of the chessboard it. In our MPI program, we generate a permutation tree that each thread handles the permutations starting with its rank. As the numbers of processors increases, each permutation subtree that starts with a value between one to Size(n), is handled by more than one processor, thus making the speed faster. And Our main process (my_rank=0) is only used to count the total number of solutions and total elapsed time.

Cores / Sizes	2	4	6	8	10
4	111	45	79	73	55
8	3467	1757	1014	557	643
10	225650	152754	73218	78288	51835

Figure 3. The elapsed time (us) based on the different number of cores and sizes in MPI

From figure 3, firstly, we can find that when size is relatively small, the difference between the sequential algorithm and the parallel algorithm is not very big. The main reason may be that the elapsed time is too short to see the difference. For instance, when size is 4, the sequential calculation time is only 26us, and the elapsed. time is very short, so when we use the parallel algorithm, the elapsed time is not significantly shortened. While when we used the MPI when size is 8 and 10, we can clearly see from the figure 3 that the elapsed time is much shorter than the sequential algorithm, and as size is larger, the elapsed time is shortened.

From the figure 3 we can also see that as the cores increases, the parallel time will be reduced accordingly. We can also see that as the cores increases, the elapsed time will be reduced accordingly. But there will be some data that is special. For example, when size is 8, the number of cores increases from 8 to 10 and the elapsed time is increased. We analyzed the reason why it is because after size equaling to 8, the processing of 8 cores has already done the number of rows for each core to process in the for loop, so after that increasing the number of cores will not actually increase the operating efficiency and reduce the time.

We also found that when size is 10, the number of cores increased from 6 to 8 but the elapsed time increased. We thought the reason is because when the core is 8, the process which n equals to 8, 9 and 10 will enter the first three processes, and the next five processes will only process once, so there is no significant improvement in efficiency.

- **OpenMP (Full permutation)**

Thirdly, we use OpenMP method to parallelize the full permutation algorithm. The experiment is based on different number of threads. The results are shown below.

Thread / Sizes	2	4	6	8	10
4	54	138	148	1725	263
8	3118	2210	1995	2771	1777
10	288631	186331	156623	117641	96394

Figure 4. The elapsed time (us) based on the different number of threads in OpenMP

As we can see from the Figure 4, when size is 4, the elapsed time for different number of threads are also quite similar to the sequential one which may be the size is too small which is not convincing that the algorithm has been parallelized successfully or not.

When size is 8, the elapsed time for different number of threads are decreased when comparing with the sequential one. However, when the total number of threads is 8, the elapsed time is increased compare to the previous 6 threads one, which may be the hardware issues because when size is equal to the total number of threads, all the threads will be used to parallelize the for loop to finding permutations. And we can see that when thread number goes to 10, the elapsed time is smaller than the 6 threads one, which prove that the 8 out of 10 threads are used. And the result of 8 threads should be considered as the occasional case.

When size is 10, the elapsed time is decreasing when the number of threads is increasing which is under expectation. And the result for 10 thread is almost 4 times faster than the sequential one, which prove that the algorithm is in a way of parallelism.

Based on our hypothesis, we try to combine the MPI with OpenMP method, but in this sequential method, since both our method is to parallel on the outermost for loop, if these two methods are used together, the final result is not executable. To achieve the merger of two parallel ideas may need to change our sequential method.

Conclusion

Through the design of this program, we have a deep understanding of several calculation methods for the N Queens Problem. More importantly, the parallel algorithm for the use of MPI and OpenMP allows us to deepen the understanding of MPI and OpenMP parallel methods.

Reference List

Aggarwal, V., Troxel, I. A., & George, A. D. (2004, September). Design and analysis of parallel N-queens on reconfigurable hardware with Handel-C and MPI. In 2004 MAPLD Intl. Conference.

Pacheco, P. (1997). Parallel programming with MPI. Morgan Kaufmann.