Report

OpenMP

Parallelization of the N Queens Problem with OpenMP was straightforward. I solve the N Queens Problem in a for loop.

```
int64_t iteration;
#pragma omp parallel for
for (iteration = 0; iteration < max_iterations; iteration++)
{
    uint64_t tag = (uint64_t)iteration;
    int i;
    int queen_rows[MAX_N];

    //generate configurations where there's only one queen per collumn
    for (i = 0; i < n; i++)
    {
        queen_rows[i] = tag % n;
        tag /= n;
    }

    if (isValid(queen_rows, n))
    {
        #pragma omp atomic
        solutions++;

        #pragma omp critical //guard lock mutex for the printing function
        write_solutions(queen_rows, i, n);
    }
}
```

In this case, my max_iterations is pow(n,n), so I basically run my brute force solving algorithm for every possible combination on the board. I wrap the for loop in a #pragma omp parallel for, and then I control the printing of the solutions to the file with a #pragma omp critical, which is essentially a mutex. The loop is executed across the number of threads that the current system has.

```
int isValid(int qr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            // two queens in the same row - return false
            if (qr[i] == qr[j]) return 0;

            // two queens in the same diagonal - return false
            if (qr[i] - qr[j] == i - j ||
                qr[i] - qr[j] == j - i)
                return 0;
        }
    }
    //otherwise return true
    return 1;
}
```

The isValid function checks whether a current configuration is a valid solution. If yes, then we print the solution and increment the total number of solutions. The increment operation itself is atomic as well.

Results vs. Serial Results:

N Value	OpenMP (seconds)	Serial (seconds)
4	0.0047017	0.0000309
5	0.0049444	0.0003386
6	0.0045355	0.0043175
7	0.0230593	0.0850285
8	0.244059	1.9522
9	5.51317	49.413
10	152.702	1375.94

From these results we can clearly see that the OpenMP implementation of the program scales much better with higher N values, however for lower N values (<6) the serial implementation is actually faster.

<thread>

Parallelization of the N Queens Problem using <thread> implementation was more complicated due to the fact that I actually had to decide and spread the work across threads myself.

```
if (argv[1] == std::string("thread")) {
    uint64_t max_iterations_absolute = 1;
    double start_time, end_time;
    int64_t solutions = 0;

    int n;
    printf("Enter the value of N: ");
    scanf_s("%d", &n);

    std::string filename = "thread_solutions_";
    filename += std::to_string(n);
    filename += ".txt";



    myfile.open(filename);

    auto threads_num = std::thread::hardware_concurrency();

    max_iterations_absolute = uint64_t(pow(n, n));

    uint64_t max_iterations_current = uint64_t(max_iterations_absolute / threads_num);

    start_time = omp_get_wtime();


    std::vector<std::thread> threads;

    int64_t iteration = 0;

    myfile << "-----------------------\n-----------------------\n   \n   \n   \n   \n   \n   \n   ";

    for (int i = 0; i < threads_num; i++) {

        threads.push_back(std::thread(solve_thread_loop, max_iterations_current, n, std::ref(solutions), iteration));
        iteration += uint64_t(max_iterations_absolute / threads_num);
        max_iterations_current += uint64_t(max_iterations_absolute / threads_num);

    }


    // Join threads
    for (auto& t : threads)
        t.join();
```

I declare 2 new variables: max_iterations_absolute and max_iterations_current. The absolute max iteration value is pow(n,n) all possible board combinations, and the max_iterations_current is the max iterations per thread. I create a thread pool, and for each thread I initialize them with their own iteration and current max iteration values. For example, lets say the absolute max number of iterations is 100 and we have a total of 10 threads. For 1st thread, we start at iteration of 0 and current max iterations of 10. The 2nd thread starts at iteration of 10, and current max iterations of 20. The 3rd thread starts at iteration of 20 and current max iterations of 30, and so on. This is how I spread the work across all my threads evenly.

```
void solve_thread_loop(int64_t max_it, int num, int64_t &solutions_thread, int64_t it) {

    for (it = it; it < max_it; it++)
    {
        uint64_t tag = uint64_t(it);
        int i;
        int queen_rows[MAX_N];

        for (i = 0; i < num; i++)
        {
            queen_rows[i] = tag % num;

            tag /= num;
        }

        if (isValid(queen_rows, num))
        {
            increment(solutions_thread);

            write_solutions_thread(queen_rows, i, num);

        }
    }
}
```

The solve_thread_loop function is just the same for loop as in the OpenMP and serial methods, except it's wrapped in a function so I can pass the whole thing to each of the threads. However, in this case it's necessary to guard lock both the total number of solutions and the function for printing out the results to the file with the mutex:

```
void increment(int64_t &sol) {
    std::lock_guard<std::mutex> lock(mut);
    sol++;
}
```

```
void write_solutions_thread(int qr[], int index, int n_number) {
    std::lock_guard<std::mutex> lock(mut);

    myfile << "\n";
    for (index = 0; index < n_number; index++)
    {
        int j;
        for (j = 0; j < n_number; j++)
        {
            if (qr[index] == j) myfile << "|Q";
            else myfile << "| ";
        }
        myfile << "|\n";
    }
    myfile << "\n";
}
```

Results vs. Serial Results:

N Value	<thread> (seconds)	Serial (seconds)
4	0.0089576	0.0000309
5	0.0093346	0.0003386
6	0.0108499	0.0043175
7	0.0230352	0.0850285
8	0.222542	1.9522
9	5.32668	49.413
10	149.659	1375.94

From these results we can once again conclude that the multithreaded version of the program scales better with higher N values, but the serial version is still faster for lower N values (<6).

Conclusion

The results from the <thread> and OpenMP implementations are extremely similar, since both are literally the same thing, just written differently. OpenMP hides the spreading of work across threads from the user and does it for us, meanwhile the user has to actually decide how to spread the work across the threads when using the <thread> implementation, however the concept is identical.

N Value	<thread> (seconds)	OpenMP (seconds)	Serial (seconds)
4	0.0089576	0.0047017	0.0000309
5	0.0093346	0.0049444	0.0003386
6	0.0108499	0.0045355	0.0043175
7	0.0230352	0.0230593	0.0850285
8	0.222542	0.244059	1.9522
9	5.32668	5.51317	49.413
10	149.659	152.702	1375.94

OpenMP does tend to have better results for the lower N values compared to <thread>. For the higher N values, <thread> seems to uphold the best results out of the 3 implementations, even compared to OpenMP. The times vary across executions, and sometimes <thread> actually gives slightly worse results compared to OpenMP and vice versa, so it's safe to assume that both are extremely similar to each other and the differences between varying results is just a margin of error. For the serial implementation, however, things get really bad the higher the value of N is, especially in this case of a brute force approach to the N Queens Problem.

Hardware setup: