Final Project

MacMillan, Kyle

December 11, 2018

Contents

List of Figures

1 Description of the program

This write-up is for the Final project and the repository for my work is here.
This program was incredibly hard until I learned of `std::next_permutation` . Until then I had a ridiculous chain of nested for loops to iterate over up to $n = 10$. Also, because it was a nested loop I had to do a horizontal check to ensure there were not duplicate queens on a line. This made the program unbearably slow. Joe informed me of `std::next_permutation` and it was a complete game changer.

Armed with this new tool I began to design a solution capable of using it. After searching around online I found a Stack Overflow post that allowed me to calculate the i^{th} permutation. Given p processors and a known number of permutations given in the form of $n!$ I can calculate the number of permutations to send to each processor. This is not the *best* load balance strategy, but it is not terrible. Since I am not pruning the permutations the only imbalance comes from transmitting that a valid board layout was found.

The graduate portion of this assignment allowed me to utilize a Task/Channel method (as described in Chapter 6) to complete the assignment. By having `MPI_Send` and `MPI_Recv` in the workers and master, respectively, I was able to meet that program requirement. Essentially the master goes directly into a receive loop, waiting for an `MPI_Send` on the appropriate tag and source. After receiving the value a counter is incremented and it checks if we have hit the expected number of solutions. If we have not finished it goes back into the receive loop, otherwise it sends out an `MPI_Bcast` to all workers that changes a boolean flag from $false$ to $true$, which stops them from checking any more boards.

1.1 Performance Analysis

Timing was performed with the simple Linux `time` command. This problem is not like our previous assignments. We have an exponential growth in the problem size *at each step*. That means we don't really need high-fidelity timing to get a picture of what's going on.

Figure 1 shows that up to $n = 11$ my sequential implementation is faster than the MPI solution shown in Figure 2. I didn't give it much thought at first and decided to try and run the problem on a "single" processor but I had left the host file filled with computers so I am not sure what the numbers in Figure 3 represent. After that blooper I created the sequential code which is a clone of the `MPI` with all `MPI` aspects stripped out.

```
7184015@linux102 seqfinal >>time ./final 11
Running 11-queens...
Found all valid queen positions: 2680

real    0m3.437s
user    0m3.436s
sys     0m0.000s
```

Figure 1: Sequential 11-queens

```
7184015@linux101 final >>time mpirun -np 129 ./final 11
Running 11-queens...
Found all valid queen positions: 2680

real    0m4.099s
user    0m0.008s
sys     0m0.020s
```

Figure 2: MPI 11-queens

```
7184015@linux102 final >>time mpirun -np 2 ./final 11
Running 11-queens...
Found all valid queen positions: 2680

real    0m6.319s
user    0m0.044s
sys     0m0.044s
```

Figure 3: "Sequential" MPI 11-queens

Figure 4 shows the sequential time for $n = 12$ and Figure 5 shows the first time the MPI solution beat the sequential solution, and it was by quite a margin.

Wanting to gather as much data as possible I ran sequential up to $n = 13$. Figure 6 shows that it took 9 minutes and 39 seconds to quit early from a sequential run of $n = 13$. Figure 7 shows it took 18 times longer to run the sequential compared to `MPI`. I calculated out the estimated sequential time for $n = 14$ at 135.1 minutes for sequential code.

```
7184015@linux102 seqfinal >>time ./final 12
Running 12-queens...
Found all valid queen positions: 14200

real    0m42.776s
user    0m42.776s
sys     0m0.000s
```

Figure 4: Sequential 12-queens

```
7184015@linux101 final >>time mpirun -np 129 ./final 12
Running 12-queens...
Found all valid queen positions: 14200

real    0m4.956s
user    0m0.020s
sys     0m0.008s
```

Figure 5: MPI 12-queens

```
7184015@linux102 seqfinal >>time ./final 13
Running 13-queens...
Found all valid queen positions: 73712

real    9m39.287s
user    9m39.268s
sys     0m0.012s
```

Figure 6: Sequential 13-queens

```
7184015@linux101 final >>time mpirun -np 129 ./final 13
Running 13-queens...
Found all valid queen positions: 73712

real    0m31.994s
user    0m0.024s
sys     0m0.024s
```

Figure 7: MPI 13-queens

2 Description of the algorithms and libraries used

The STL's `next_permutation` was a keystone in the design and execution of this program. Also used was `MPI`,

3 Description of functions and program structure

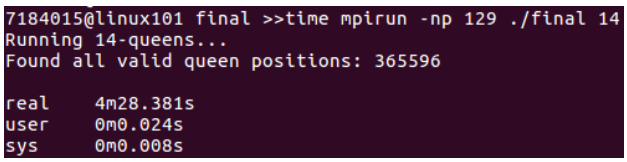

Figure 8: MPI 14-queens

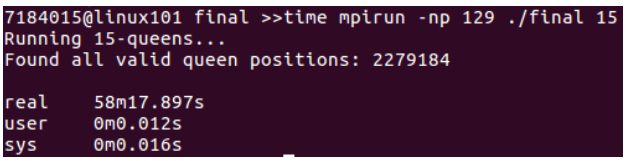

Figure 9: MPI 15-queens

4 How to compile and use the program

4.1 MPI Run

This program can be compiled with the Makefile. From the 'final' folder simply type:

```
make
```

or

```
make final
```

To use the program type:

```
mpirun -np 129 .\final 13
```

or

```
mpirun -np 129 .\final 13 printout
```

'129' represents the number of processors and '13' represents the n part of the n-queens problem. Master node $id = 0$, therefore it is necessary to run with `-np 2` or more. It **will not run properly** if you specify `-np 1`. If you wish to run it sequentially see Section 4.2. The `printout` must follow the integer and is used to print valid queen positions to the console if you wish to have printouts.

4.2 Sequential

This program can be compiled with the Makefile. From the 'seqfinal' folder simply type:

```
make
```

or

```
make final
```

To use the program type:

```
.\final 10
```

or

```
.\final 10 printout
```

10 can be any integer, though I do not recommend going above 12 because run time is likely to exceed 9 minutes as shown in Figure 6. The `printout` must follow the integer and is used to print valid queen positions to the console if you wish to have printouts.

5 Description of the testing and verification process

Testing was done with printout verification and count verification. What that means is I bounced the printout arrays of n-queen positions and verified them by hand. Figure 10 shows an example printout. This could obviously only be done on smaller n values. For larger n values I assumed the diagonal check was correct (since it was verified good at the lower level) and only verified with the count method. A function was made to test the count against a const array of known solution for a given n.

```
Running 5-queens...
Arrangement: {0, 2, 4, 1, 3} PASSED
Arrangement: {0, 3, 1, 4, 2} PASSED
Arrangement: {1, 3, 0, 2, 4} PASSED
Arrangement: {1, 4, 2, 0, 3} PASSED
Arrangement: {2, 0, 3, 1, 4} PASSED
Arrangement: {2, 4, 1, 3, 0} PASSED
Arrangement: {3, 0, 2, 4, 1} PASSED
Arrangement: {3, 1, 4, 2, 0} PASSED
Arrangement: {4, 1, 3, 0, 2} PASSED
Arrangement: {4, 2, 0, 3, 1} PASSED
Found all valid queen positions: 10
```

Figure 10: Testing n-queens output

6 Description of what you have submitted

Included in the submission is the code needed to compile the program, a Makefile to compile said code, and a detailed write-up of the assignment in pdf form.

7 Additional Thoughts

I really wish we had more time for this project. I would have really liked to dig into `MPI` more; it's extremely powerful. I wanted to go about this problem in a totally different route but I was time constrained (due to coursework and personal reasons).

Ideally I believe this problem would be solved with an `MPI_Send` and `MPI_Recv` in each worker. Instead of splitting the entire set of permutations amongst p processors I wanted to split *half* of the permutations amongst p processors. Then when one finished early through *pruning* it could push an `MPI_Send` call to the Master node, at which point it would `MPI_Recv` the request and `MPI_Send` a new permutation along with k permutations to run through. What this would have accomplished was essentially a worker queue so as workers finished they would request more work. Given that this problem has very dynamic work loads this would be a "near-ideal" plan.

I wanted to do pruning but it added more complexity than I was willing to commit to with the time available. It'd be really neat to have a "Distributed Computing" or "Parallel 2" course that built on `CUDA` and `MPI`.

Overall fun course and I may be able to use `MPI` for my thesis work.