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 and 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.

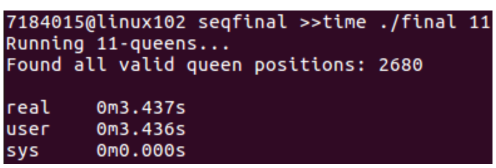

Figure 1: Sequential 11-queens

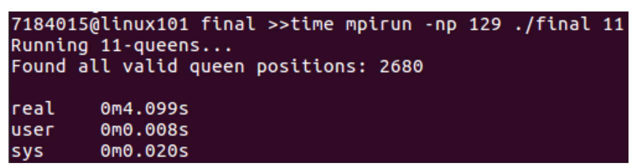

Figure 2: MPI 11-queens

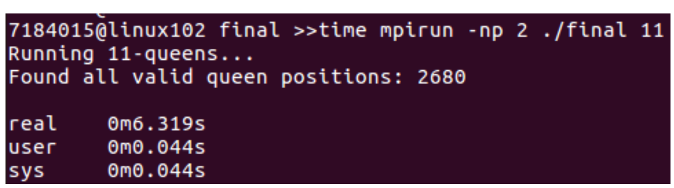

Figure 3: "Sequential" MPI 11-queens

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.

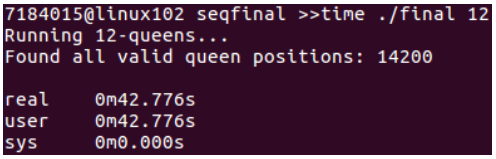

Figure 4: Sequential 12-queens

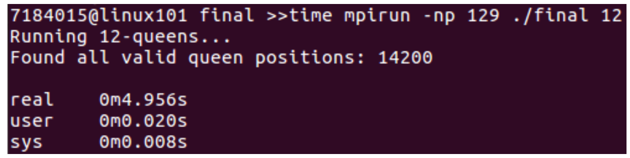

Figure 5: MPI 12-queens

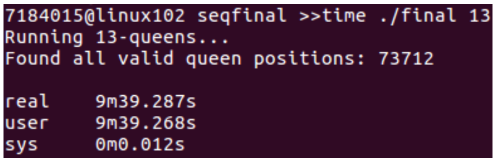

Figure 6: Sequential 13-queens

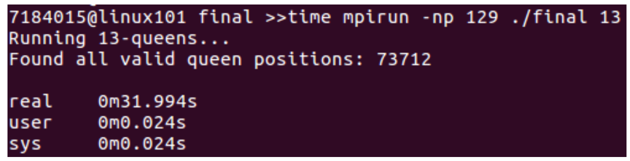

Figure 7: MPI 13-queens

Given all of this data I can run some performance metrics on it. Lets start with an observation/estimation. As can be seen in Figure 10 it takes approximately 3 seconds to run a solution that takes 0.005 seconds when ran sequentially. We can therefore assume a base overhead $\kappa = 3$ seconds. This number likely increases as more `MPI_Send` and `MPI_Recv` calls are made. Ψ (speedup) is variable in this case because the amount of work being done.

$$\Psi = \frac{\sigma(n)}{\varphi(n)}$$

So in the case of $n = 12$ we see a speedup of:

$$\Psi = \frac{42.776}{4.956} = 8.631154157$$

So in the case of $n = 13$ we see a speedup of:

$$\Psi = \frac{579.287}{31.994} = 18.106113646$$

These are simple speedup values, but an 18 times increase is amazing. I did not run sequential at $n = 14$ because the math showed it would take around two hours to run. If we assume 2 hours for that and the time shown in Figure 8 of 4 minutes and 28 seconds we get:

$$\Psi = \frac{7200}{268.381} = 26.8275325$$

and for $n = 15$:

$$\Psi = \frac{121650}{3497.897} = 34.778039491$$

A 34 times speedup, and it will only increase from there. As we can see there is a trend to this speedup but it is going to cap eventually. The reason for that is because we are not increasing the number of processors as the amount of work we have to do is increased.

Next lets calculate efficiency which is:

$$Efficiency = \frac{Sequential\ Execution\ Time}{Parallel\ Execution\ Time * Processors}$$

$$\epsilon_{12} = \frac{42.776}{4.956 * 129} = 0.066908172$$

$$\epsilon_{13} = \frac{579.287}{31.994 * 129} = 0.14035747$$

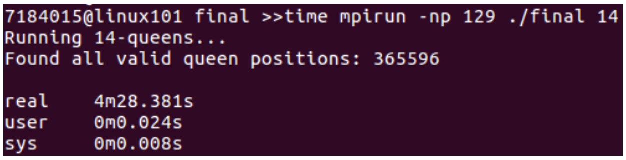

Figure 8: MPI 14-queens

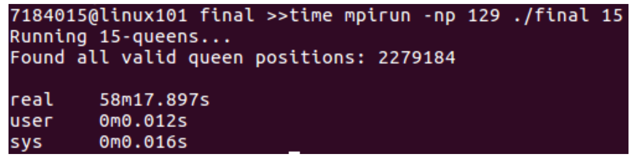

Figure 9: MPI 15-queens

$$\epsilon_{14} = \frac{7200}{268.381 * 129} = 0.207965368$$

So we can see efficiency continue to increase as the problem size grows. I expect it to take approximately 121,650 seconds (or 33 hours) to run sequential $n = 15$. Using the timing from Figure 9 we can estimate efficiency:

$$\epsilon_{15} = \frac{121650}{3497.897 * 129} = 0.269597205$$

The efficiency increase is approximately linear, which is interesting.

Moving on, the Karp-Flatt metric is defined as:

$$f_e = \frac{\frac{1}{\Psi(n,p)} - \frac{1}{p}}{1 - \frac{1}{p}}$$

We can take our experimentally gained speedups and apply them to this metric:

$$f_{12} = \frac{\frac{1}{8.631154157} - \frac{1}{129}}{1 - \frac{1}{129}} = 0.108952012$$

$$f_{13} = \frac{\frac{1}{18.106113646} - \frac{1}{129}}{1 - \frac{1}{129}} = 0.047848948$$

$$f_{14} = \frac{\frac{1}{26.8275325} - \frac{1}{129}}{1 - \frac{1}{129}} = 0.029753851$$

$$f_{15} = \frac{\frac{1}{34.778039491} - \frac{1}{129}}{1 - \frac{1}{129}} = 0.021165916$$

So rather than calculate the efficiency of throwing additional processors at this problem we calculated what kind of efficiencies we could expect as our problem continued to grow exponentially. The linear-scaling efficiency paints a picture that we will eventually reach maximum efficiency for this system, at which point we would need to increase the processors to see any more gain. The predicted maximum efficiency is around $n = 26$, but that doesn't take into account the fact that I used $uint32\_t$ instead of $uint64\_t$. 64 bit integers tend to take longer to work with.

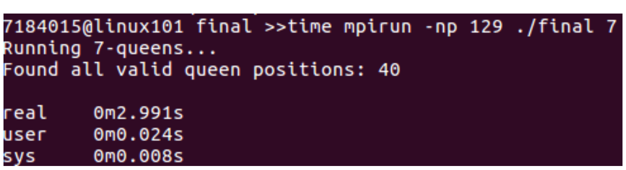

Figure 10: MPI 7-queens

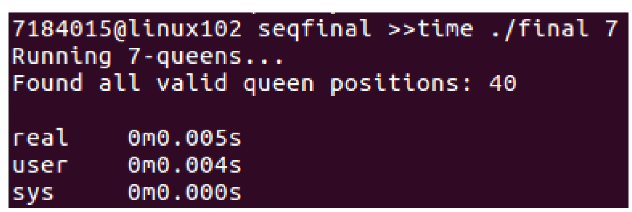

Figure 11: Sequential 7-queens

2 Description of the algorithms and libraries used

The STL's `next_permutation` was a keystone in the design and execution of this program. `MPI` was used. The Stack Overflow post for n^{th} permutation was incredibly useful.

3 Description of functions and program structure

3.1 Program Structure

This program was setup with a Master/Slave (aka worker) configuration. The Master in this case was `linux102` or `linux103` and the workers were `linux01-16`. I love classes so I wrote this in `C++` and included a Board class. Essentially when the program fires up it verifies user input and then spins up the Master/Worker relations. Master eagerly awaits `MPI_Send` commands from the workers and then updates the "found" count. Once the count reaches the threshold for this n-queens problem the Master uses `MPI_Bcast` to change a boolean flag which causes the workers to shutdown.

There is a "feature" I left in the code that causes crashes. I tried everything I could think of but due to the graduate requirement of early exit it completely messes with the `MPI_Send` and `MPI_Recv`. I tried all kinds of combinations to get it to exit cleanly but if you put too many processors on a small problem it will hang for a minute then dump. My `MPI_Send` was only being called on success. That leaves the Master hanging in a receive loop.

Sequential works a little different in that instead of a Master/Worker relationship it has these 8 lines of code:

```
uint8_t *queens = (uint8_t*)malloc(n * sizeof(uint8_t));
for (uint8_t i = 0; i < n; ++i){
    queens[i] = i;
}
std::cout << "Running " << int(n) << "-queens..." << std::endl;
SBoard b(factorials[n], n, print_out, queens);
b.validSBoardPermutations();
free(queens);
```

There are two helper `.h` files: `completion.h` and `nthpermutation.h`. The helper files contain constants and functions relevant to completion of tasks as well as the code to calculate the n^{th} permutation.

3.2 Description of Functions

Each function has a header if you wish to know details. Please see my repository here and here for `MPI` and sequential solutions respectively.

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. I send the valid queen positions to console and verified them by hand. Figure 12 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 and only verified with the count method. A correctCount function was made to test the count against a const array of known solutions 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 12: Testing n-queens output

6 Description of what you have submitted

Included in the submission is the code needed to compile the program, two Makefiles 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.

8 Additional Timings

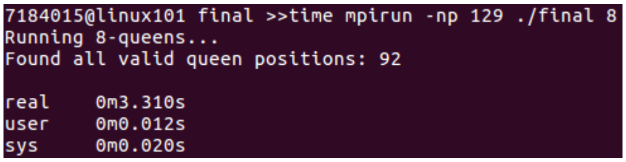

Figure 13: MPI 8-queens

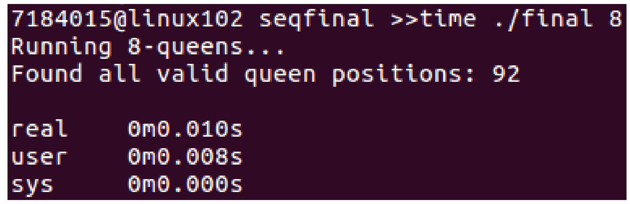

Figure 14: Sequential 8-queens

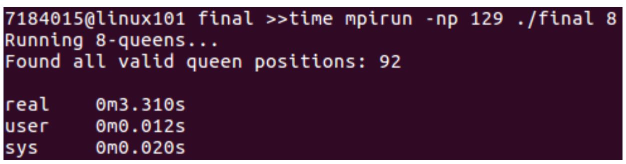

Figure 15: MPI 8-queens

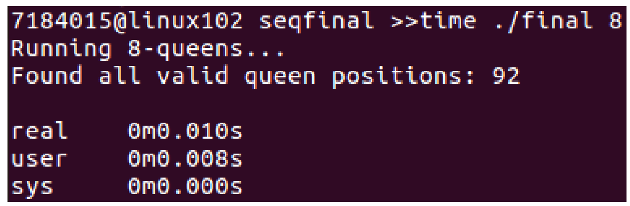

Figure 16: Sequential 8-queens

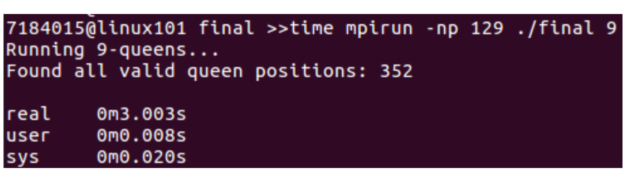

Figure 17: MPI 9-queens

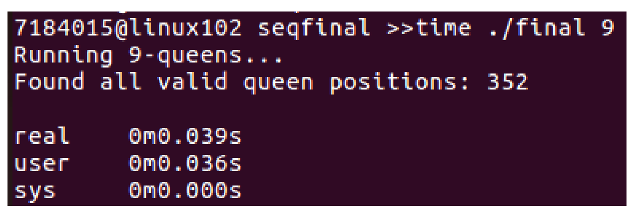

Figure 18: Sequential 9-queens

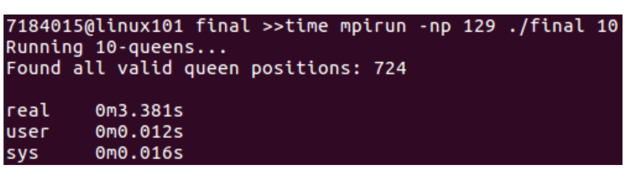

Figure 19: MPI 10-queens

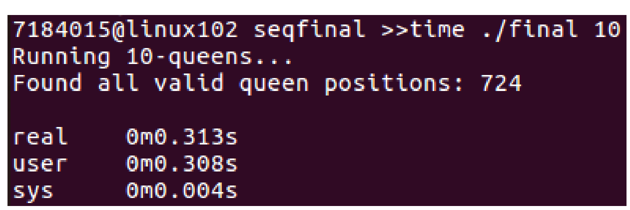

Figure 20: Sequential 10-queens