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.

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

4 How to compile and use the program

This program can be compiled with the Makefile. Simply type:

```
make
```

or

```
make final
```

To use the program type:

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

'129' represents the number of processors and '13' represents the n part of the n-queens problem. This was coded to deny use of less than a $1:1$ **ratio of processors to boards**. It was completely possible to code it up to accept less but increased complexity unnecessarily. Essentially if the number following np is greater than the number following `.\final` taken to the factorial **it will not run**.

Example: `mpirun -np 12 .\final 3!` fails because 12 is greater than 3!.

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. 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.

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.