PROJECT PROPOSAL PHPC-2017

A High Performance Implementation of the N-Queens Solver: An MPI Approach to the Backtracking Algorithm

Principal investigator (PI)	Alexander Lorkowski
Institution	EPFL-CSE
Address	Route Cantonale, CH-1015 LAUSANNE
Involved researchers	Only PI
Date of submission	May 1, 2017
Expected end of project	June 2, 2017
Target machine	DENEB
Proposed acronym	NQUEEN

Abstract

The brute-force solution of N-Queens problem is computationally expensive as the number of possible positioning of the n number of queens on an nxn chessboard is defined as $(n^2!)/(n!(n^2-n)!)$[1]. In order to meet this challenge, we present a fully functional parallel implementation of the N-Queens solver based on the backtracking algorithm. The goal of this research project is to demonstrate the advantage of parallel implementation for the given problem, and also reveal the potential pitfalls and restrictions one can encounter. Our application is coded in C++ and uses the industrial standard MPI[2].

Code availabe at:
https://c4science.ch/diffusion/3874/phpc-nqueenmpi.git
ssh://git@c4science.ch/diffusion/3874/phpc-nqueenmpi.git

1 Scientific Background

The N-Queens problem requires solving the placement n number of queens on an nxn chessboard in a way that no two queens can attack each other. In other words, no two queens can be on the same row, column, or diagonal. Practical applications of the N-Queens problem include parallel memory storage schemes[3] and traffic control[4]. It is also similar to problems that examine permutations of original configurations of information such as the traveling salesman problem.

A parallel implementation for solving the N-Queens problem on a distributed system using Message Passing Interface (MPI) is presented. This method is based on the backtracking algorithm where in which we divided the problem into smaller subproblems specified by the number of processors available.

2 Implementations

The application is implemented in C++. We have investigated an implementation of the N-Queens solver using a distributed memory version using the MPI library[2]. The code has been fully debugged using the gdb debugger. The Valgrind tool has been used to remove all the memory leaks.

2.1 The backtracking algorithm

Common implementations of the backtracking algorithm use a solution matrix that takes O(N^2) space per solution. The implementation reported here uses O(N) space per solution by storing the solution in an one dimensional array which we store the placement of the queen at the i-th row in the j-th index of the array. The backtracking algorithm first places a queen on the first column and stores the index of the row into the array. The algorithm traverses the chessboard by column, then by row, adding to the array and recursively calling itself as long as no conflict occurs. The conflict is defined as the following:

Algorithm 1 Conflict Check

```
1: if (Chessboard[j] == i or abs(Chessboard[j] - i) == ColumnIndex - j) then return True
```

The pseudocode of the backtracking algorithm is provided below. We note that this pseudocode only provides the backtracking component. The full NQueen function includes a section to write out the solution to a file.

Algorithm 2 Backtracking N-Queen Algorithm(currentColumnIndex, rowOfQueenInCurrentColumn, numberOfQueens, nextColumnIndex, Chessboard, Solution[n], numberOfSolutions)

```
1: for i = 0 to numberOfQueens to place do
2:     if j is less than currentColumnIndex and !conflict then return j = j + 1
3:     Chessboard[nextColumnIndex] = i
4:     CurrentColumnIndex = nextColumnIndex
5:     nextColumnIndex = nextColumnIndex + 1
6:     recursive calls to Backtracking N-Queen Algorithm(arguments...)
```

If a conflict occurs, the algorithm attempts to put the next queen on the next row or if all remaining placements are explored, the algorithm returns back one element in the history array to continue traversing for the solutions, hence backtracking. Once the history array contains N elements, the algorithm has obtained one solution to the problem.

2.2 MPI Implementation

Within the MPI communicator, processes are identified by their rank within the communicator. This property allows the implementation of the master-slave scheme which follows the procedure outlined below:

1. One node is designated as the master node. All other nodes are used as Slaves. Communication between them is done via MPI calls.

2. In the process of solving the N-Queen problem, each Slave signals to the Master that it is idle, and the Master sends out instructions and data to the Slaves. The instruction and data are smaller parts of the whole that are handled internally by the Slave nodes.

3. Any Slave node works through its task and once it is finished, it signals to the Master requesting another set of instruction. This is executed in the same manner until a stop signal is received, at which point execution ends.

4. The Master continues to distribute tasks to the Slaves until all instructions have been communicated, by which point it sends a stop signal to all Slave processes.

Regarding step 2, the total problem is divided into a number of subproblems depending on the number of running processors. If the number of processors is less than the number of queens, the number of subproblems is equal to the number of queens; otherwise, the problem is subdivided based on two columns, increasing the number of sub problems to (N-1)(N-2). Minimizing the difference between the number of (N-1)(N-2) subproblems and the number of processors result in faster runtimes.

As an additional comparison, another MPI program that does not employ the master-slave scheme is provided. In this program, each processor independently selects a sub problem based on its processor rank.

The MPI program solves for the total number of solutions. The solutions can be printed on console using the -p flag. Additionally, the -u flag can be used to solve for number of unique solutions as well as the -g flag that prints a game board representation rather than an array of solutions. Provide -h for more options.

3 Architecture

We ran our production version on the Deneb cluster at EPFL. This machine presents the following characteristics[7]:

- Peak performance: 293 TFLOPs (211 in CPUs, 92 in GPUs)
- Storage: 350TB
- 376 compute nodes, each with

  2 Ivy Bridge processors running at 2.6 GHz, with 8 cores each,

  64 GB of DDR3 RAM,
- 144 compute nodes, each with

  2 Haswell processors running at 2.5 GHz, with 12 cores each

  64 GB of DDR4 RAM
- Infiniband QDR 2:1 connectivity,
- GPFS filesystem.

4 Theoretical Scaling Analysis

Profiling of the serial code was done through the use of gprof. The profiling is done with the -f flag. This flag indicates to the program to solve the solutions using symmetry operations. As such, it is the fastest setting of the program. The profiling for two problem sizes is shown in Table 1. The second entry of the flat profile for each problem size is the allocation of a C++ vector for the storage of data. Unfortunately, the this allocation cannot be parallelized. In result, our code appears only 75% parallelizable.

Table 1: The three longest calls of the running program profilied using gprof for finding all solutions to problem using symmetry (-f) of sizes 11 and 15.

Number of Queens	%Time	Cumulative Seconds	Self Seconds	Name
N = 11	40.00	0.04	0.04	SparseNQueenSolver::solve()
	25.00	0.07	0.03	std::vector int, std::allocator int
	10.00	0.08	0.01	Chessboard::getState()
N = 15	32.10	71.33	71.33	SparseNQueenSolver::solve()
	31.34	141.15	69.82	std::vector int, std::allocator int
	11.87	167.51	26.36	Chessboard::setState()

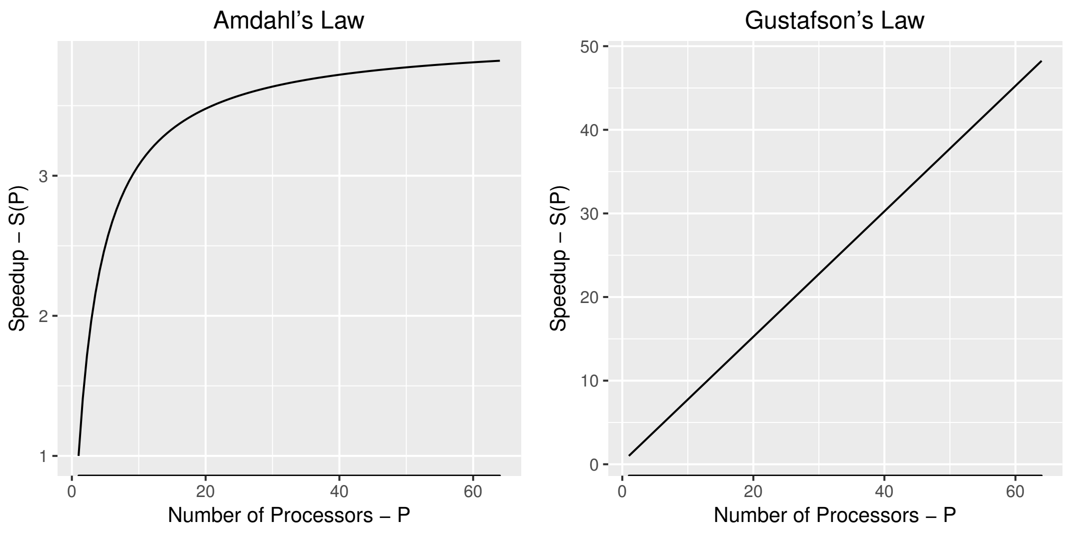

Figure 1: (Right) Strong scaling of the NQueen solver and (left) weak scaling of the NQueen solver.

The serial, non-parallelizable part of the algorithm is about 25% to 30%. Figure 1 depicts the theoretical strong scaling of the application based on Amdahl's law.The weak scaling was calculated using Gustafson's law using the same profiling information from the strong scaling.

5 Benchmarking

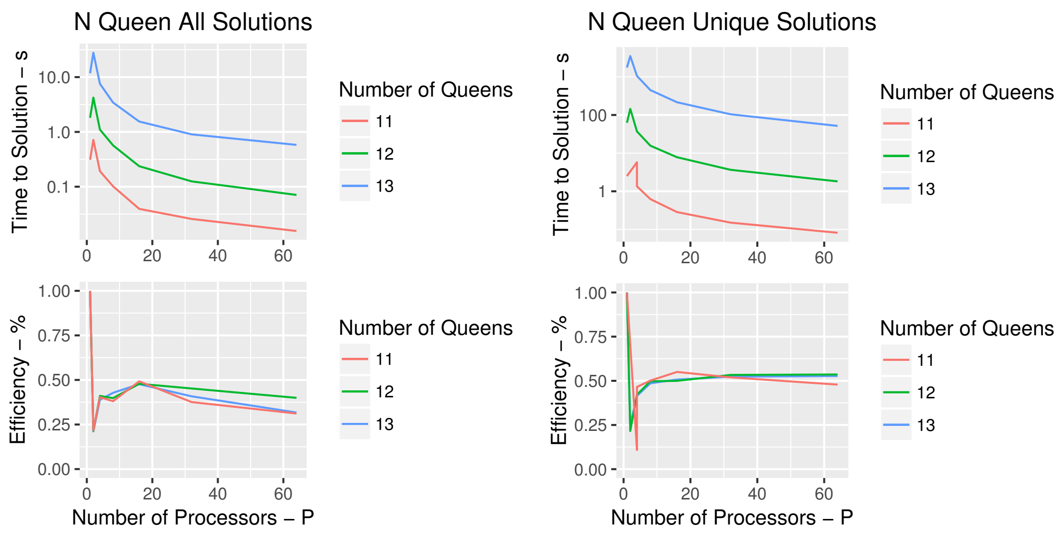

Figure 2: (Right) Time to solution under the master-slave scheme for solving all possible configurations and (left) for all unique configurations.

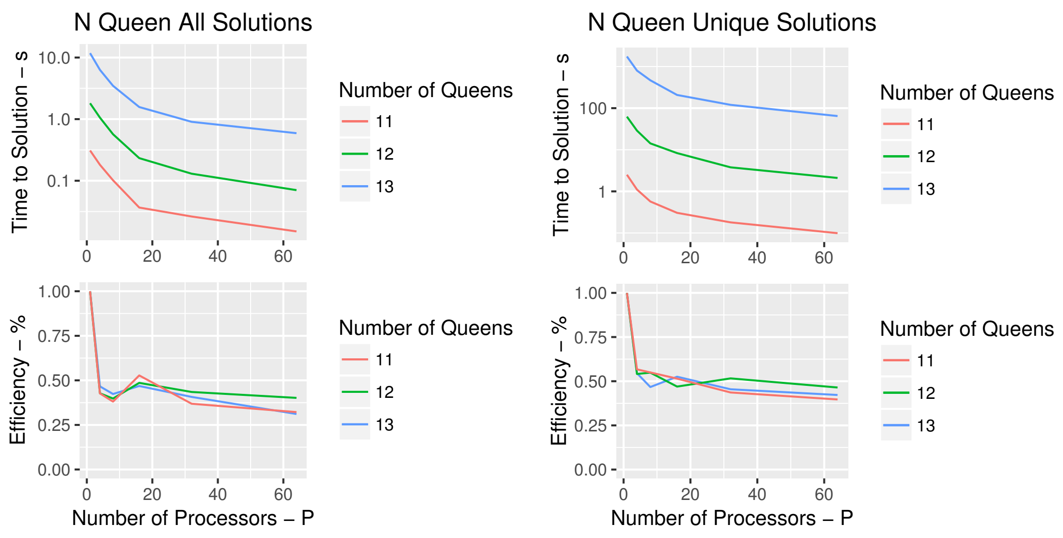

Figure 3: (Right) Time to solution not under the master-slave scheme for solving all possible configurations and (left) for all unique configurations.

Benchmarking was done with relatively large problem sizes for the N-Queen problem. Unfortunately, the total number of solutions to the N-Queen problem grows exponentially. This property limits our current capability to asses the code for larger problem sizes.

The benchmark test provide an illustration of the benefits of extra processors in Figure 5. There is a notable spike in the time to solution for 2 processors and this is more evident in the efficiency plot where the efficiency starts at approximately 50%. This is due to the Master-Slave scheme implemented in the parallel code. Running the parallel code with two processors is essentially running the serial code with the addition of MPI communication overhead. An increase in the number of processors mitigates this effect in what appears to peak in efficiency at approximately 18 processors.

Efficiency when specifying the -u flag to solve for all unique solutions is low for every problem size and number of processor combination. This is expected as the number of solutions that the code has to sort thorugh exponentially increases and is not helped by the fact that the algortihm for checking is computationally intensive. It is important to note that the code block for extracting the unique solutions does not follow the Master-Slave scheme and together with the fact that this is a major bottleneck of the code, is the reason why we do not see a well defined peak as seen in the efficiency of Figure 5 (Right).

Figure 5 show the benchmarking of the MPI code that does not employ the master-slave scheme. There are performance gains over the master-slave MPI protocol discussed above for both small problem sizes and few total number of processors. However, this implementation plateaus in efficiency at around 50% much like the master-slave scheme for high processor counts. Furthermore, larger problem sizes may cause load balance issues. Because each processor is not assigned a subproblem, but rather independently determines the subproblem to solve itself, it is likely that one processor under unfortunate circumstance always chooses subproblems that have the most work and may lead other processors to be idle.

6 Resources budget

In order to fulfill the requirements of the project, we present hereafter the resource budget.

6.1 Computing Power

The parallel code will be implemented and optimized using the N-Queens problem size of N=12 or N=16. This will allow development of the code for a reasonably large problem size within a reasonable time. Due to the nature of this problem, a small increase in the problem size can result in a significant increase in the number of solutions, hence an increase in calculations and need for memory. Therefore, it is anticipated that a minimum of 10 GB will suffice for this project; however, a maximum total memory of 100 GB will allow for testing and devlopment of the parallel code past the size of the problem demonstrated in this proposal ($N > 12$).

6.2 Raw storage

We anticipate that a temporary disk space of 100 GB for a single run and a total permanent disk space of 1 TB will suffice for this project. The space complexity of the code is $\theta(N)$ per solution if no flags are specified. If the -p flag is specified, the code will produce a file that requires $\theta(N^2)$ space per solution.

The test case used in this report (N=12) only outputs 5.8 MB of data; however, increasing the size to N=16 results in approximately 1000-fold increase in the number of solutions, in which we expect the amount of storage required to scale the same. In order to test the code on the larger N-Queens simulations, we will require the raw storage specified.

6.3 Grand Total

Total number of requested cores	`64 [cores]`
Minimum total memory	`10 [GB]`
Maximum total memory	`100 [GB]`
Temporary disk space for a single run	`100 [GB]`
Permanent disk space for the entire project	`1 [TB]`
Communications	`Pure MPI`
License	own code (BSD)
Code publicly available ?	`Yes`
Library requirements	`MPI`
Architectures where code ran	`Intel 64, Deneb`

7 Scientific outcome

The N-Queens solver can be efficiently implemented in parallel. Despite the increased speeds that can be obtained through parallelization, the N-Queens probelm remains computationally intensive and expensive particularly for large N. Futher optimization of the MPI code is proposed such as implementing an OpenMM and MPI hybrid approach or subdividing the problems more efficently using a divide-and-conquer approach.

References

[1] J. J. Watkins, *Across the Board: The Mathematics of Chessboard Problems*, Princeton University Press, 2007, ISBN 0-69-113062-0

[2] The MPI Forum, *MPI: A Message-Passing Interface Standard*, Technical Report, 1994

[3] C. Erbas, and M.M. Tanik, *Storage schemes for parallel memory systems and the n-queens problem*, The 15th ASME ETCE Conference, Computer Applications Symposium, Houston, Texas, pp. 115-120, January 26-30 1992

[4] R. Sosic , J. Gu, *A polynomial time algorithm for the N- Queens problem*, ACM SIGART Bulletin, Volume 1, Number 4, pp.7-11, October 1990

[5] Rolfe, Timothy J., *A Specimen MPI Application: N-Queens in Parallel*, inroads SIGCSE Bulletin, Vol. 40, No. 4, 2008

[6] A. Khademzadeh, M.A. Sharbaf, and A. Bayati *An Optimized MPI-Based Approach for Solving the N-Queens Problem*, The Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Victoria, Canada, pp. 119-124, November 12-14, 2012

[7] Scientific IT and Application Support, `http://scitas.epfl.ch`, 2016

[8] Virtual Institute - High Productivity Supercomputing, `http://www.vi-hps.org`, 2016

[9] Scalasca, `http://www.scalasca.org`, 2016