COMP 529 TERM PROJECT

Multithreaded N-queens problem with OpenMP and MPI

HANDAN KILINÇ

03.01.2013

1. Abstract

The N-queens problem has gained much attention over the past century. In this project, genentic algorithm approache was adopted to tackle this problem. Also, a paralel genetic algorithm is demonstrated as a possible way to increase genetic algorithm speed with using mpi and openmpi. Results are shown in graphics and analyzed.

2. Inroduction

There is a class of problems in computer science where the search space is very large and the search path does not need to be remembered. These problems cannot be solved by a standard search such as a breadth-first searched, an algorithm guaranteed to find solution, because the search space is not computable in polynomial time. To solve these problems in a reasonable time, heuristic methods must be used. Genetic algorithms are powerful heuristic methods, capable of searching large spaces of possible solutions. However, due to intense computations performed by genetic algorithms, some form of parallelization is desirable to increase performance. [1]

2.1 N-Queen problem

The n-queens problem is the problem of placing n queens on an n x n chessboard so that no two ttack, i.e., so that no two are in the same row, column or diagonal. Thus, solutions to this problem can be represented by n x n permutation matrices, i.e., matrices of zeros and ones in which there are exactly one 1 in every column and every row. [2]

2.2 Genetic Algorithm

A genetic algorithm begins with a set of randomly produced states (each state is an individual) known as the population. If a solution to the problem is contained within the initial state population, then the algorithm is complete; otherwise, the algorithm must continue. The production of the population’s next generation is based on several functions: fitness, selection, crossover, and mutation. This process continues until either a predetermined limit (usually time or number of generations) is reached or a solution is produced. The most important components of genetic algorithms are:

- **Representation (definition of individuals)** : Objects forming possible solution within original problem context are called genotypes. Chromosome and individual can be used for points in the genotype space. Elements of a chromosome are called genes.

- Evaluation **function (or fitness function):** This function calculates the fitness value which shows how the chromosome closes to the solution.
- **Population:** The role of the population is to hold possible solutions. A population is a multiset of genotypes.
- **Parent selection mechanism:** The role of parent selection (mating selection) is to distinguish among individuals based on their quality to allow the better individuals to become parents of the next generation.
- **Variation operators (crossover and mutation):** Mutation is applied to one genotype and delivers a modified *mutant*, the *child* or *offspring* of it. In general, mutation is supposed to cause a random unbiased change. A binary variation operator is called *recombination* or *crossover.* This operator merges information from two parent genotypes into one or two offspring genotypes.

Genetic algorithm is step by step following [4]:

1. Create a population of *k* individuals through random generation of *k* states.
2. Evaluate fitness of each individual within the population and assign a higher value to those with a “better” state of fitness.
3. Select mating pairs (parents) at random with regard to each individual’s fitness proportion as measured across the entire population.
4. Cross-over genetic material from each pair of mates by randomly selecting crossover points and swapping genetics from each parent into their offspring (the next generation).
5. Mutate the offspring as determined by a small independent probability that some of its genetic material will be altered.
6. Continue to evaluate fitness for each generation of the population until either a solution or cut-off limit is realized.

3. Serial Genetic Algorithm of Nqueen

The classic combinatorial problem is to place eight queens on a chessboard so that no two attack. This problem can be generalized as placing *n* nonattacking queens on an *n´n* chessboard. Since each queen must be on a different row and column, we can assume that queen *i* is placed in *i*-th column. All solutions to the *n*-queens problem can therefore berepresented a s *n*-tuples (*q1*, *q2*, …, *qn*) that are permutations of an *n*-tuple (1, 2, 3, …, *n*). Position of a number in the tuple represents queen's column position, while its value represents queen's row position (counting from the bottom) Using this representation, the solution space where two of the constraints (row and column conflicts) are allready satistfied should be searched in order to eliminate the diagonal conflicts. Complexity of this problem is *O(n!)*. Figure 1ilustrates two 4-tuples for the 4-queen problem. [1]

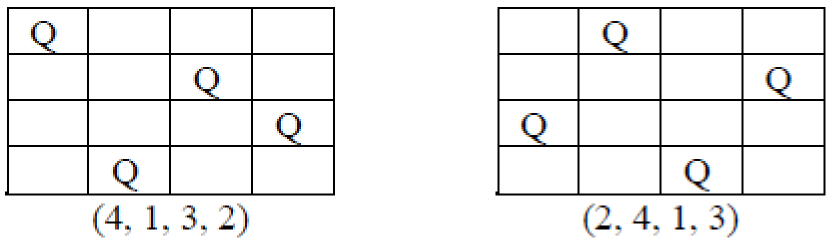


Figure 1: *n*-tuple notation examples

To understand the solution is right or wrong, there is fitness value which shows how we close to the solution. Since *n*-tuple representation eliminates row and column conflicts, wrong solutions have queens attacking each other diagonally. A fitness function can be designed to count diagonal conflicts: more conflicts there are, worse the solution. For a correct solution, the function will return zero.

3.1 Initialize Population

To initialize population, firstly a random and appropriate (which does not have same numbers) chromosome is created. After, as swapping this chromosome's random two elements, new chromosomes are added into population until population has enough chromosome.

3.2 Fitness Function

For a simple method of finding conflicts [3], consider an *n*tuple: (q_1,..., q_i,..., q_j, ..., q_n). *i*-th and *j*-th queen share a diagonal if:

$$i - q_i = j - q_j$$

or

$$i + q_i = j + q_j$$

which equals:

$$||q_i - q_j|| = || i - j||$$

If we use this approach to find fitness value, we have to look every couple and so the complexity is *O(n^2)*. In [1], there is a solution to find fitness in *O(n)*. Therefore, the [1] fitness function is used in this project. Pseudo code of the fitness function is following:

```
set left and right diagonal counters to 0

for i= 1 to n
   left_diagonal[i+$q_i$]++
   right_diagonal[n-i+$q_i$]++
end
sum = 0
for i = 1 to (2n-1)
   counter = 0
   if (left_diagonal[i] > 1)
      counter += left_diagonal[i]-1
   if (right_diagonal[i] > 1)
      counter += right_diagonal[i]-1
   sum += counter / (n-abs(i-n))
end
```

3.3 Mutation operator

It chooses a randomly chromosome from the population. After, two position is randomly selected and swapped the numbers. It creates a new individual which is similar the original one.

3.4 Tournament selection

When selection part happens to select two parents for crossover, choosing good parents (better fitness) is important. If we choose two best fitness parents, two find these parents takes time since selection part happens a lot of time. Therefore, we first choose randomly five chromosome and then we chooses two best fitness parents.

3.5 Crossover

In our serial genetic algorithm, we use partially matched crossover. First step is random selection of two positions within chromosomes and exchange of genetic material. In most cases, this will result in invalid tuples, since numbers in a tuple must be unique. Second step in PMX crossover eliminates duplicates.

3.6 General Algorithm

1. Initialize population (until population size is equal START_SIZE)
2. Create new generation (until population reaches MAX_POPULATION)
   - Crossover
3. Make new initialization. Go back number 2.

4. Parallel Genetic Algorithm of Nqueen

4.1 OpenMp Approach

In this approach, the loops which take long time paralyzed. When program makes crossover with PMX, eliminating duplicate genes has nested loops. Also, when population is renewing, all of the pointers which are represented as chromosomes are deleting in the memory and it needs parallelization. Besides, fitness function need parallelization when n- queen size is big.

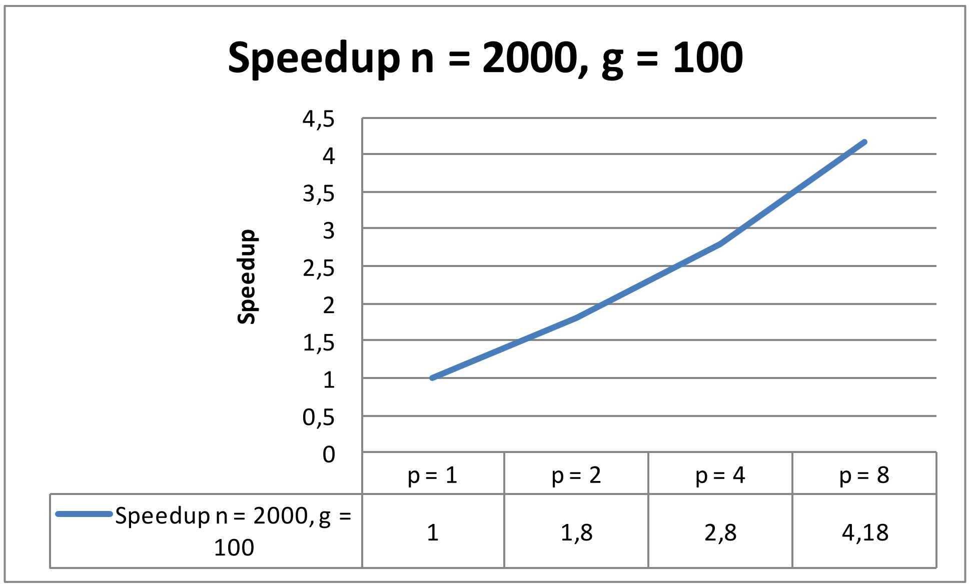
According to these observation, "#pragma omp parallel for" added in these loops. Parallelization runs only when the loop size is bigger than granularity, otherwise serial for loop is running.

4.1.1 Graphs and Analyze

When speedup calculates, instead of comparing elapsed time of parallel and serial code, we use tour completion time, because genetic algorithm base on randomness to calculate elapsed time. 1 tour completion speed shows that the code will create every tour in this speed. In one tour, new generation is created and all crossover, mutation and fitness functions are in the tour. So, faster tour is faster solution.

For example, in serial code 1 tour takes 2 seconds and in parallel code it takes 1 second. Lets say, after 10 tour serial code found a solution, so elapsed time is about 20. In this case if parallel code found 25 tour, its elapsed time is 25 which is bigger than serial code. However, parallel is faster because it can find in the same randomness with serial code in 10 sec.

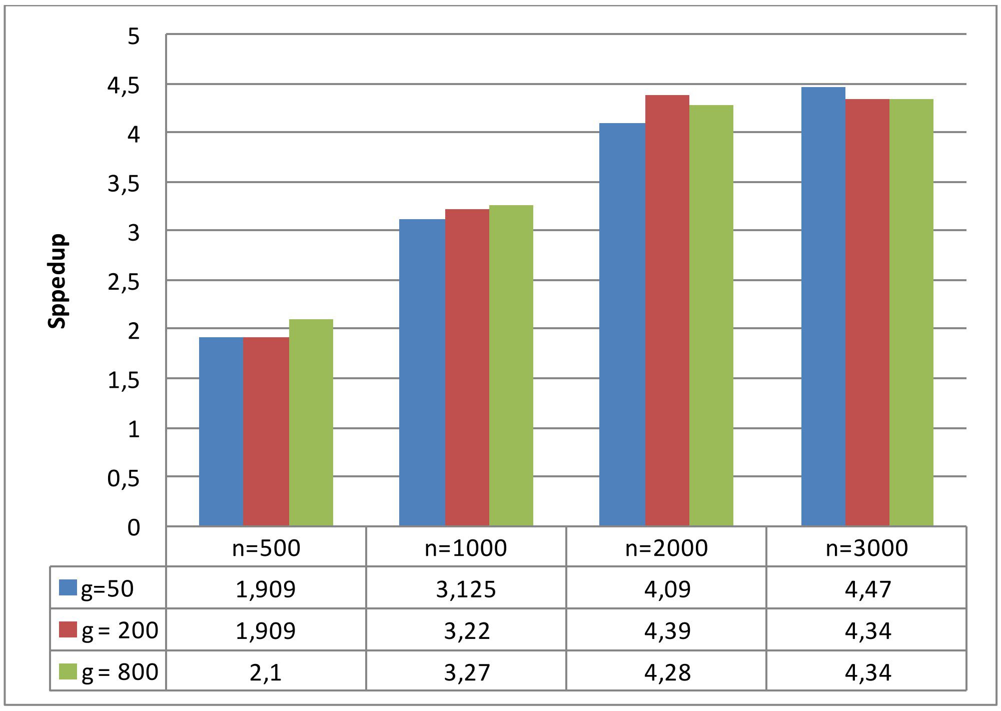
In following graphs p is thread number, n is queen number and g is granularity.



	p = 1	p = 2	p = 4	p = 8
Speedup n = 2000, g = 100	1	1,8	2,8	4,18

Graph 1: Speed up n = 2000, g = 100

In Graph 1, speed up is increasing with the thread number. In p = 2, the speed up nearly 2, but in other speed ups there is not linear increasing with thread number because of the overhead.



	n=500	n=1000	n=2000	n=3000
🟦 g=50	1,909	3,125	4,09	4,47
🟥 g = 200	1,909	3,22	4,39	4,34
🟩 g = 800	2,1	3,27	4,28	4,34

Graph 2: According to g and n, when p = 8

The speed ups are nearly equal because in some loops (like findDiffrence() method) to make parallel for is randomly because the loop size chosen randomly and sometimes it is smaller than g and sometimes not. From the graph, we can say that when g is increases in n = 2000 an n=3000 because of the overhead speedup is increasing. From overall graph, we can see that there is a n increase with n. It is reasonable, because when the parallel algorithm gives better result when n is bigger than 1000. In small size, overhead prevent more speedup.

N	Time (sec)	Tour Num
10	0.038	16. tour
30	2.02	802. tour
100	2.85	3762. tour
500	144.48	13265. tour
1000	636.669	100969. tour

Table 1: N queen solution time in p = 8 g= 100

4.2 MPI Approach

In mpi approach, master and slaves are used. The main idea is to distribute expensive tasks across slaves (controlled by a master process) to be executed in parallel [1]. Master assigns a part of population to each slave and waits for them to finish. In the "createNextGenearation" method, slaves creates new child and send them to master. Master adds the child that he receives. And he also makes mutation which is easy job.

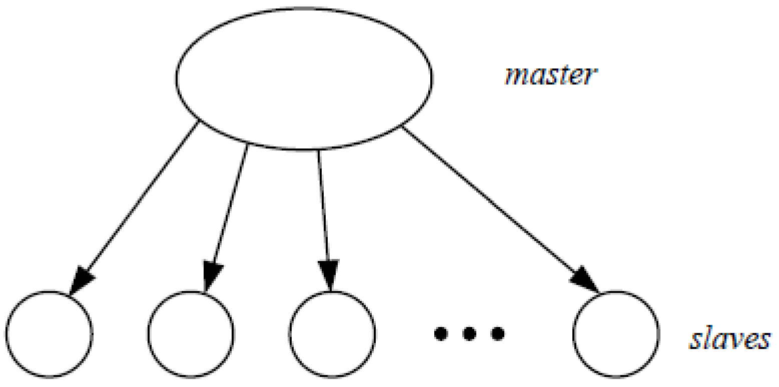


Figure 2: Basic structure of mpi code

The following code implements this idea [Geneticmpi.cpp]

```
MPI_Init(NULL, NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
float popDivision = float(MAX_POPULATION - START_SIZE) / float(comm_size * 2);

MPI_Status status;
if(my_rank != 0){
    int loop = int(popDivision);
    for(int i = 0; i < loop; i++){
        int* mates = makeTournament();
        vector<int*> children = Crossover(population[mates[0]], population[mates[1]]);
        MPI_Send(children[0], SIZE, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
        MPI_Send(children[1], SIZE, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
    }
}else{
    int loop = popDivision * comm_size;
    for(int i = 0; i < loop; i++){
        Chromosome child1, child2;
        MPI_recv(child, SIZE, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
        child1 = child;
        MPI_recv(child, SIZE, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
        child1 = child;
        addNewIndividual(child1);
        addNewIndividual(child2);
        if(count % 20 == 0){
            mutRound = MUTATION_RATE * popSIZE;
            for(int i = 0; i < (int) mutRound; i++){
                randIndx = rand() % (popSIZE - 1);
                mutation(population[randIndx]);
            }
        }
        count++;
    }
```

As a result, expensive part of the code is executed by slaves and they do not have to wait anything therefore, we expect that speed up is more than openMp version of the code. However, there will be communication overhead between master and slave. So we cannot wait linear speed up.

Conclusion

GAs' have provided significant improvements in terms of efficiency when compared to search algorithms such as depth-first search with backtracking. The reasons for the gains in efficiency are two-fold [5]. First, GAs' utilize implicit parallelism, i.e. maintaining different candidate solutions in parallel. Second, the greediness of heuristics can be offset by randomly generating offspring (candidate solutions) [4]. We get good speed ups in openMp (also expected in mpi) in large problem size. In other search algorithms, to find a solution in reasonable time even in parallel programming is hard. Also, genetic algorithm finds most of the time fastest solution.

References

[1] Solving n-Queen problem using global parallel genetic algorithm, Marko Božikovic, Marin Golub, Leo Budin
[2] **A Parallel Algorithm for the n-Queens Problem,** Lorna E. Salaman Jorge

[3] Ellis Horowitz and Sartaj Sahni, *Fundamentals of computer algorithms*, Computer Science Press Inc., Rockville, MD, 1978.
[4] **Solvıng N-Queens With A GeneticAlgorithim,** *Richard Eastridge and Cecil Schmidt Washburn University*