EUROCON 2003 Ljubljana, Slovenia

Solving n-Queen problem using global parallel genetic algorithm

Marko Božiković, Marin Golub, Leo Budin
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
marko.bozikovic@alterbox.net, marin.golub@fer.hr, leo.budin@fer.hr

***Abstract*--This paper shows a way that genetic algorithms can be used to solve n-Queen problem. Custom chromosome representation, evaluation function and genetic operators are presented. Also, a global parallel genetic algorithm is demonstrated as a possible way to increase GA speed. Results are shown for several large values of n and several conclusions are drawn about solving NP problems with genetic algorithms.**

***Index Terms*--global parallel genetic algorithm, n-queen problem, tournament selection.**

I. INTRODUCTION

PROBLEMS with no deterministic solutions that run in polynomial time are called NP-class problems. Because of their high complexity (e.g. $O(2^n)$ or $O(n!)$) they cannot be solved in a realistic timeframe using deterministic techniques. To solve these problems in a reasonable amount of time, heuristic methods must be used.

Genetic algorithms (GAs) are powerful heuristic methods, capable of efficiently searching large spaces of possible solutions. However, due to intense computations performed by GAs, some form of parallelization is desirable to increase performance.

This paper will present an implementation of global parallel GA for solving n-Queen problem.

II. N-QUEEN PROBLEM

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 $n \times n$ 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 be represented as n-tuples (q_1, q_2, ..., q_n) 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 1 ilustrates two 4-tuples for the 4-queen problem.

Q			
		Q	
			Q
	Q		

(4, 1, 3, 2)

	Q		
			Q
Q			
		Q	

(2, 4, 1, 3)

Figure 1: n-tuple notation examples

The problem with determining a good fitness function for n-Queen problem is the same as for any combinatory problem: the solution is either right or wrong. Thus, a fitness function must be able to determine how close a wrong solution is to a correct one. 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.

For a simple method of finding conflicts [4], consider an n-tuple: (q_1,..., q_i,..., q_j, ..., q_n). i-th and j-th queen share a diagonal if:

$$i - q_i = j - q_j \tag{1}$$

or

$$i + q_i = j + q_j \tag{2}$$

which reduces to:

$$\left\| q_i - q_j \right\| = \left\| i - j \right\| \tag{3}$$

This simple approach results in fitness function with complexity of $O(n^2)$. It is possible to reduce complexity to $O(n)$ by observing diagonals on the board. There are $2n$-1 "left" (top-down, left to right) and $2n$-1 "right" (bottom-up, right to left) diagonals (see figures 2 and 3)

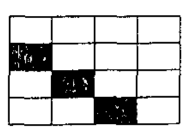

Figure 2: Third "left" diagonal

Figure 3: Second "right" diagonal

A queen that occupies i-th column and q_i-th row is located on $i+q_i-1$ left and $n-i+q_i$ right diagonal. A fitness function first allocates counters for all diagonals. Then, for each queen, counters for one left and one right diagonal that queen occupies are increased by one. After evaluation, if a counter has a value greater than 1, there are conflicts on the corresponding diagonal. Fitness value is obtained by adding counter values decreased by 1 (except for counters with value 0) 4 shows a pseudocode for such a function. Note that each counter value is normalized with respect to length of corresponding diagonals.

```
set left and right diagonal counters to 0

for i= 1 to n
   left_diagonal[i+qi]++
   right_diagonal[n-i+qi]++
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
```

Figure 4: Fitness function for n-queen problem

III. GENETIC ALGORITHMS

Genetic algorithms are search and optimization procedures based on 3 biological principles: selection, crossover and mutation. Potential solutions are represented as individuals that are evaluated using a fitness function representing a problem being optimized. Basic structure of a genetic algorithm is shown in the following list:

1. A random population of individuals (potential solutions) is created. All individuals are evaluated using a fitness function.
2. Certain number of individuals that will survive into next generation is selected using selection operator. Selection is somewhat biased, favoring "better" individuals.
3. Selected individuals act as parents that are combined using crossover operator to create children.
4. A mutation operator is applied on new individuals. It randomly changes few individuals (mutation probability is usually low)
5. Children are also evaluated. Together with parents they form the next generation.

Steps 2.-5. are repeated until a given number of iterations have been run, solution improvement rate falls below some threshold, or some other stop condition has been satisfied.

One modification of this basic structure is a 3-way tournament selection used here. Instead of selecting individuals from one generation to the next, selection and crossover are performed continuously. First, 3 individuals are selected completely at random. Then, two individuals with the highest fitness are combined using crossover to produce an offspring that will replace the worst individual. There is no clear distinction between generations.

Individual representation and fitness function for n-Queen problem were presented in the previous chapter. It is also necessary to design proper crossover and mutation operators that will operate on n-tuple representation.

Mutation operator used is very simple: for a given tuple, we randomly select two positions and swap the numbers. This creates a new individual, similar to the original one, and validity of the tuple is preserved. An example is given in figure 5:

(5 1 3 2 7 4 6)
Becomes
(5 7 3 2 1 4 6)

Figure 5: Mutation operator

There are several possibilities for a crossover operator. First version is equivalent to the mutation operator: swapping two random positions in a tuple. Obvious drawback of this operator is that it does not combine genetic material of parents.

Another crossover operator is PMX[1] crossover. It is similar to two-point binary crossover operator. First step is random selection of two positions within chromosomes and exchange of genetic material:

(2 5 1 | 3 8 4 | 7 6)
(8 4 7 | 2 6 1 | 3 5)
Becomes
(2 5 1 | 2 6 1 | 7 6)
(8 4 7 | 3 8 4 | 3 5)

Figure 6: PMX crossover – first step

In most cases, this will result in invalid tuples, since numbers in a tuple must be unique. Second step in PMX crossover eliminates duplicates. In the example above, number 2 occurs at positions 1 and 4 in the first offspring. The 2 at position 4 is newer (from the crossover), so the 2 at position 1 is changed into 3 that was at position 4 before the crossover.

[1] Partially Matched Crossover

(2 5 1 | 3 8 4 | 7 6)
(8 4 7 | 2 6 1 | 3 5)
becomes .
(3 5 4 | 2 6 1 | 7 8)
(6 1 7 | 3 8 4 | 2 5)

Figure 7: PMX crossover – second step

The third operator is designed for 3-way tournament selection: parents are compared, and equivalent positions are copied to the offspring. Other positions in the offspring tuple are filled in randomly, but care is taken to preserve tuple validity. If parents are equivalent, one of them is replaced by a randomly created tuple to avoid chromosome duplication.

(2 5 1 3 8 4 7 6)
(2 4 7 3 6 1 8 5)
(2 8 6 3 1 4 7 5)

Figure 8: 3-way tournament crossover

IV. Global parallel genetic algorithm

For solving n-Queen problem, a Global Parallel Genetic Algorithm (GPGA) was used. Figure 9 shows basic structure of a GPGA:

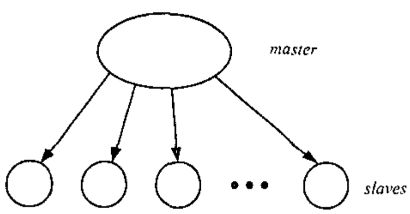

Figure 9: Basic structure of a GPGA

The main idea is to distribute expensive tasks across slaves (controlled by a master process) to be executed in parallel. In a classic configuration, the master maintains a population and executes genetic operators (selection, crossover and mutation), while slaves perform evaluation. Master assigns a part of population to each slave and waits for them to finish[2]. GPGA can achieve significant increase in speed, especially for expensive evaluation functions or large populations. However, due to communication between the master and slaves, there is an upper limit for the number of slave processes. Further speed gains are limited by master-slave communication overhead.

A slightly modified configuration was used for solving n-Queen problem. Since 3-way tournament selection is used as a selection operator, several slaves can run tournament selection and crossover in parallel, while master performs only mutation. Figure 10 shows master pseudocode, and figure 11 shows slave pseudocode:

```
create initial population
evaluate initial population

create slaves

while not done
   start slaves
   wait for slaves to finish
   run mutation operator
end
```

Figure 10: Master pseudocode

```
for i = 1 to slave_iterations
   select 3 individuals
   run crossover operator
   evaluate offspring
   if solution found set done=true;
end
```

Figure 11: Slave pseudocode

V. Experiments and results

For testing purposes, a custom C++ program was written. The master and each of the slaves are run in separate threads, so the program can be executed on a multi-processor machine for full speed benefits. Also, since each thread keeps track of its own running time, the program can be used to simulate a multi-processor execution on a single-processor machine.

Experiments showed that PMX [3] and 3-way tournament crossover operators don't behave well, since they tend to generate close solutions rather quickly, but fail to produce correct solutions in a reasonable amount of time. They also tend to unify the solution pool, so the only force of change in a GA run becomes the mutation operator.

As a final crossover operator, simple mutation operator was used, slightly modified to fit 3-way tournament selection. After selection, evaluation and comparison, one of the two surviving individuals is selected at random and a mutated copy is used to replace the third individual.

Convergence rate and speed of the algorithm were greatly improved this way. Also, convergence rate was much more uniform across runs, obtaining a solution for a given size of the problem within a rather narrow range of iterations. All results presented here were obtained using mutation as a crossover operator.

Problem sizes used were 100, 200, 500, 1000 and 2000 queens. For all problem sizes, two slaves were running 100 iterations per one master iteration each, with population sizes of 100 individuals, and mutation rate was 0.02.

Ten runs were performed for each problem size, and average results are given in the table 1. The machine used was a P4@2.4GHz, running Windows™ XP Professional.

[2] Synchronous GPGA. In an *asynchronous* GPGA, the master continues with execution while slaves are running.

Queens	I_M	T_M	T_{S1}	T_{S2}
100	537	0.015	0.547	0.540
200	1346	0.062	1.435	1.447
500	6073	0.553	24.74	24.74
1000	11395	1.96	93.18	93.30
2000	26132	8.7	433.5	433.7
I_M - total number of master iterations				
T_M - total master running time (sec)				
T_{S1} - total slave 1 running time (sec)				
T_{S2} - total slave 2 running time (sec)				

Table 1: Average values of test runs

Table 2 shows results for 1000-queen problem with different number of slaves. Since these experiments were just simulations on the same single-processor P4@2.4GHz machine, the values differ from results that would be obtained on a real multi-processor system. Still, even the simulation clearly shows increase in execution speed. Each table entry represents average values for 5 runs.

Slaves	I_M	T_M	T_S
1	17236	2.967	142.1
2	9198	1.595	75.20
3	5770	1.300	65.10
4	4457	0.794	40.51
I_M - total number of master iterations			
T_M - total master running time (sec)			
T_S - average slaves running time (sec)			

Table 2: GPGA execution times for different number of slaves

VI. Conclusion

This paper showed that *n*-Queen problem can be successfully solved using genetic algorithms. Although *n*-Queen problem does not have much practical use, it represents a large class of NP problems that cannot be solved in a reasonable amount of time using deterministic methods.

Although they were conceived as heuristic methods for solving problems with "better" and "worse" solutions, genetic algorithms proved able to solve combinatory problems with simple "yes" and "no" answers. Furthermore, tests showed that GA is able to find different solutions for a given number of queens.

Since GAs perform large number of computations, parallelization can significantly improve their performance. One parallelization scheme, a global parallel genetic algorithm (GPGA) was presented here. 3-way tournament selection enabled slaves to run simultaneous selections and crossovers, freeing master process from most tasks (population initialization and mutations during the run were still performed by the master thread) GPGA is not suitable for massive parallel processing, but it shows increase in performance for a small number of parallel-processing units.

To obtain expected nearly linear increase in computation speed, experiments should be performed on a real multi-processor system, since thread context switching influences results during simulation on a single-processor machine

Appendix A: 500-Queen Solution

137	90	153	300	413	154	460	419	116	426	332	322
129	182	155	125	273	189	307	132	334	326	193	255
459	403	9	243	183	367	414	156	26	430	393	395
385	144	192	226	346	317	333	88	69	237	486	355
284	170	279	97	293	268	336	342	59	100	303	201
405	245	311	203	80	161	195	17	412	445	330	191
169	283	257	474	262	331	25	421	286	123	434	439
104	340	401	359	101	351	278	148	488	428	377	381
219	497	259	358	224	173	397	75	43	451	66	118
301	202	119	57	343	94	46	12	93	260	418	467
197	478	130	287	113	288	458	249	479	234	171	146
362	236	319	269	111	218	32	205	391	491	246	71
469	423	274	121	267	185	73	384	196	214	42	50
37	124	406	127	199	396	472	141	425	220	296	315
48	242	337	47	470	206	379	492	294	20	471	347
14	261	56	139	2	338	38	86	39	304	432	394
8	372	422	96	67	5	354	462	477	117	172	33
27	89	230	265	493	107	447	126	10	82	106	241
41	435	109	145	499	128	480	285	498	490	321	465
16	276	496	4	484	410	35	187	72	476	388	398
158	320	13	357	248	436	281	49	375	142	29	61
91	235	365	399	290	390	51	473	15	387	427	482
222	54	166	335	60	204	415	190	0	475	227	411
63	328	21	487	392	143	258	120	115	442	468	256
160	162	368	22	291	345	84	325	382	221	212	186
244	440	402	373	327	370	83	314	176	270	70	223
299	369	36	240	250	371	400	95	494	452	79	3
135	344	420	443	24	1	165	352	364	179	7	444
389	275	122	200	112	431	103	252	62	310	305	277
99	23	323	302	138	30	446	64	31	215	356	178
378	77	433	380	105	429	441	114	297	456	360	188
198	58	18	466	404	28	228	217	247	147	163	177
34	271	92	461	53	438	164	489	233	495	207	134
481	424	231	383	306	208	180	308	167	353	44	324
408	110	318	289	376	253	416	463	225	81	272	45
55	157	108	181	386	152	78	329	457	409	363	65
238	213	313	348	131	312	136	52	254	140	194	11
149	437	417	68	102	211	292	266	464	448	374	133
159	361	407	450	175	6	282	366	449	309	298	453
150	264	239	87	263	339	74	454	168	483	232	216
485	210	341	251	40	174	76	184	316	98	19	85
295	350	151	229	349	280	455	209				

References

[1] David E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley Publishing Company Inc., Reading, MA, 1989.

[2] M. Golub, D. Jakobovic, *A new model of global parallel genetic algorithm*, Proceedings of the 22nd International Conference ITI2000, Pula, 2000, pp. 363-368.

[3] Kelly D. Crawford, *Solving n-Queen problem using genetic algorithms*, Tulsa University

[4] Ellis Horowitz and Sartaj Sahni, *Fundamentals of computer algorithms*, Computer Science Press Inc., Rockville, MD, 1978.

[5] Eric Cantú-Paz, *A summary of research on parallel genetic algorithms*, Computer Science Department and The Illinois Genetic Algorithms Laboratory (IlliGAL), University of Illinois at Urbana-Champaign, cantupaz@uiuc.edu

[6] Eric Cantú-Paz, *A survey of parallel genetic algorithms*, Computer Science Department and The Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, cantupaz@illigal.ge.uiuc.edu