Concurrent Computation for Genetic Algorithms

Kittisak Kerdprasop and Nittaya Kerdprasop

Data Engineering Research Unit, School of Computer Engineering,
Suranaree University of Technology, 111 University Avenue,
Nakhon Ratchasima 30000, Thailand
{kerdpras, nittaya}@sut.ac.th

Abstract. Genetic algorithms are kind of metaheuristic method in that they apply evolutionary techniques (best-fit selection, crossover, and mutation) to find optimal solution by iteratively attempting to improve current candidate solution with respect to the fitness criteria. The use of heuristic-based search through the application of reproduction, recombination, and mutation mechanisms keeps genetic algorithms from exploring the entire search space, and thus converging to the best solution quickly. We study the mechanisms of genetic algorithms and suggest that they can perform the search procedure more quickly with the concurrent programming paradigm. In this paper, we present the implementation of concurrent genetic algorithms with Erlang, which is the powerful functional language that provides message passing features for concurrent processing. Source codes for a simple mathematical problem using genetic algorithms are provided in the paper as well as the running results. The experimental study confirms the computational time efficiency of our concurrent genetic algorithms comparative to the sequential coding style.

Keywords: Concurrency, Genetic Algorithms, Erlang Language, Functional Programming, Message Passing.

1 Introduction

Evolution in the nature has inspired several computational models including genetic algorithms, genetic programming, and evolutionary programming to solve search and optimization problems. Genetic algorithms have been successfully applied to solve planning and parameter tuning problems in manufacturing, computational sciences, mathematics, business, and many other fields. The success of genetic algorithms is due to their simple procedure of evolving successive generations of individuals to quickly converging the search strategy to the best solution that has been encoded as individuals in the population.

We investigate the robust search technique of genetic algorithms and propose that the algorithms can be improved via concurrency. In this paper, we introduce a simple model of concurrent genetic algorithms and show their effectiveness in terms of a manageable size of program source code and computational time. Our implementation is based on the concept of functional programming that provides a declarative style of coding and also a set of features for handling concurrent processing.

2 Preliminaries and Related Work

Genetic algorithms are search and optimization methods inspired by the natural selection process that causes biological evolution [7]. At the initial stage, genetic algorithms model a population of individuals by encoding each individual as a string of alphabets called a chromosome. Some of these individuals are possible solutions to a problem. To find good solution quickly, the algorithms emulate the strategy of nature, that is, survival of the fittest. Individuals that are more fit, as measured by the fitness function, are more likely to be selected for reproduction and recombination to create new chromosomes representing the next generation. Reproduction and recombination are normally achieved through the probabilistic selection mechanism together with the crossover and mutation operators.

As a consequence of their simple and yet effective search procedure, genetic algorithms have been successfully applied to solve different kinds of work ranging from optimization problem in large building structures [1], telecommunication routing protocol design [6], concurrent engineering for manufacturing design [2], to the data structure design [11] and deadlock detection [3]. Parallel computation for genetic algorithms has been proposed for at least two decades to speedup the computational time. Cantu-Paz [5] proposed the Markov chain models for parallelization of genetic algorithms. Sehitoglu and Ucoluk [9] proposed parallelization at a fine grain level of chromosome bits. Lim and colleagues [8] parallelized genetic algorithms using grid computing. The recent work of Tagawa [10] presented his study on the concurrent differential evolution using multi-core architecture. Our work presented in this paper propose a simpler scheme toward high performance computing using message passing mechanism, instead of a more sophisticated techniques appeared in the literature. The work of Bienz et al [4] is close to ours, but their process interaction scheme is more tightly coupled than our scheme.

3 Functional Programming to the Implementation of GAs

The implementation of genetic algorithms uses a simple mathematical problem: find the maximum squared number of an integer from the search space of mixed positive integers ranging from 1 to 16,777,127. The correct solution is 281,472,426,579,600. Main module of our program is the function go() that takes three parameters, that is, the population size, probability of mutation, and probability of crossover. Program source code in Erlang is given as follows:

```
go(PS,PM,PC)-> p([max_is, max()]),  Popu = init(PS, space()),
                evol(PS, PM, PC, Popu, maxLoop(),false).
max()-> round(math:pow(2,bit())-1).
bit()-> 24 . % 24=2**24   instances including 0
maxLoop()->150.   correct()-> 0.99999.
space()-> lists:seq(1,round(math:pow(2,bit())-1) ).
init(PS,L)->   random:seed(erlang:now()), Pop = randW(L,PS) ,
               lists:map(fun encode/1,Pop).
randW(_,0)-> [];      % random polulation with replacement
randW(L,N)-> [lists:nth(random:uniform(length(L)),L ) |  randW(L,N-1)].
evol(PS,PM,PC,Popu,0,_)-> p([in_each_evol,hd(Popu)]),hd(Popu) ;
evol(PS,PM,PC,Popu,_,true)-> p([in_each_evol2,hd(Popu)]),hd(Popu) ;
evol(PS,PM,PC,Popu,Max,false)->
```

```
      PopuNew = xover(PM,PC,Popu)++Popu, Lout = sel(PS,PopuNew),
      [{Tmp,_,_}|_] = Lout, Percent=Tmp/max(),
      p([after_evol_loop , maxLoop()-Max+1,max,Tmp/max()]),
      OverThresh = Percent>correct(),
      evol(PS,PM,PC,Lout,Max-1,OverThresh).
sel(PS,Popu)->   %select good  parent
      Lsort=lists:sort(fun ({_,_,X},{_,_,Y})-> X>Y  end, Popu),
      {L1,_}=lists:split(PS,Lsort) , L1 . % select best rank
xover(PM,PC,[])-> [];
xover(PM,PC,[X1,X2|T])-> xv(X1,X2,maybe(PC),PM)++xover(PM,PC,T).
xv(X1,X2,false,PM)-> [X1,X2];  % no crossover
xv(X1,X2,true,PM)-> cross(X1,X2,PM) . %  crossover
cross({_,X1,_},{_,X2,_},PM)->
       Rand=random:uniform(length(X1))-1,
       {L1,L11}= lists:split(Rand,X1), {L2,L22}= lists:split(Rand,X2),
       Xnew1= mutString(L1++L22,PM),
       Xnew2= mutString(L2++L11,PM), % mutate
       V1= decode(Xnew1,bit()),V2= decode(Xnew2,bit()),
       [{V1,Xnew1,fitness(V1)},{V2,Xnew2,fitness(V2)}].
mutString([],PM)->[];
mutString([H|T],PM)->Prob=maybe(PM) ,
       if Prob->  [(H+1) rem 2 |mutString(T,PM) ]; % mutate
          true -> [H |mutString(T,PM) ]  % no mutate
       end.
maybe(Prob)-> random:uniform() < Prob.
encode(N)-> { N , bitOf(N,bit()),fitness(N) }.
decode([],_)->0;
decode([H|T],B)-> round(H*math:pow(2,B-1))+decode(T,B-1).
bitOf(_,0)->[];
bitOf(N,B)-> bitOf(N div 2,B-1)++[N rem 2].
fitness(A)->   A*A .
p(L)-> lists:foreach(fun(H)->io:format("~p ",[H])end,L),io:format("~n").
```

4 Concurrent Genetic Algorithms Implemented with Erlang

On the design of concurrent computation (Figure 1), we try to keep the message communication as simple as possible. The main process simply creates the child process and waits for the first best result to arrive. As soon as the main process receives the first solution, it will kill other processes that are still active. This problem has only one best solution, so we accept the first one. Implementation of this concurrent scheme is in Figure 2, whereas its running result is illustrated in Figure 3.

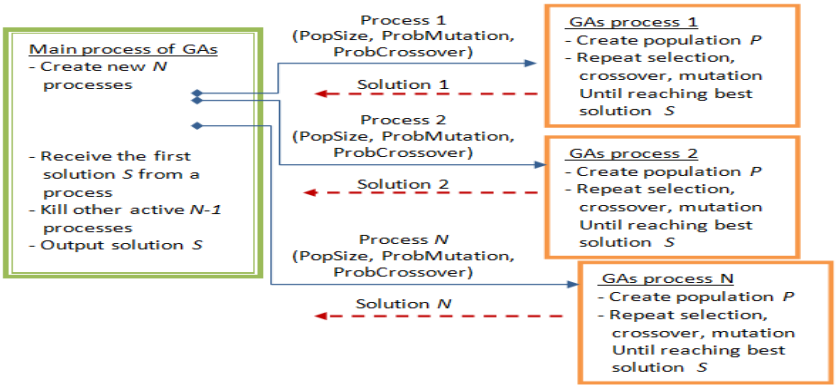

Fig. 1. A model of message passing in concurrent genetic algorithms.

```
-module(ga).    % concurrent processes
-compile(export_all).
main([P1,M1,C1],[P2,M2,C2]) ->
       Pid2 = spawn(?MODULE, process, [P1,M1,C1]),
       Pid3 = spawn(?MODULE, process, [P2,M2,C2]),
             %parameter: PopSize, ProbMutation, ProbCrossover
       Pid2 ! {self()},
       Pid3 ! {self()},
       p([all_pid,Pid2,Pid3]),
       receive
             {Pid, Msg} ->  io:format("P ~w  Value=~p~n",[Pid,Msg]),
                            exit(Pid2,kill),exit(Pid3,kill)
       end.
 process(PS,PM,PC) ->  R = go(PS,PM,PC),
                       receive {From}-> From ! {self(), R} end.
```

Fig. 2. Concurrent genetic algorithms with two active processes in Erlang.

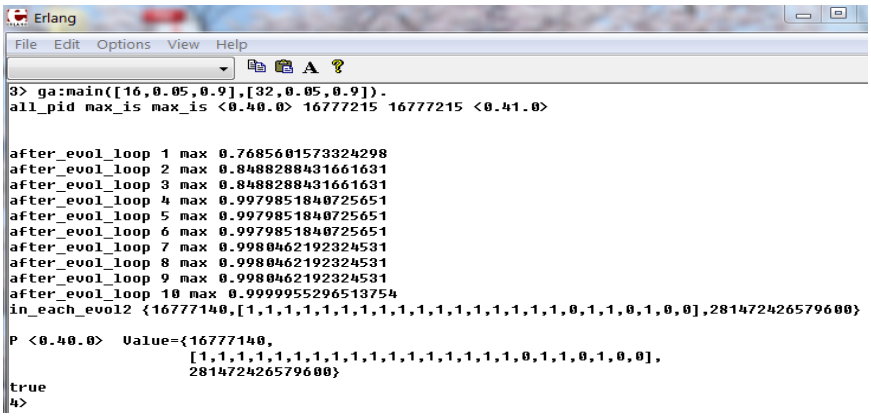

Fig. 3. Running result of concurrent genetic algorithms with two processes. The first process (process-id = <0.40.0>) has population size = 16, probability of mutation = 0.05, and probability of crossover = 0.9. The second process (process-id = <0.41.0>) has population size = 32, the other two parameters are the same as the first process.

5 Performance Evaluation

We design a series of experimentation to compare performance of sequential genetic algorithms against the concurrent implementation. The problem domain is the same as those demonstrated in Sections 3 and 4. The number of processes in the concurrent implementation has been varied from 2, 4, 8, 16, 32, 64, and 128. When the number of processes has been increased to 256, memory capacity is not enough for the Erlang system to reach the completion stage. If we, however, decrease the problem domain, the Erlang system can spawn more than hundreds of processes.

To record running time of genetic algorithms, we use the following commands:

```
f(),T1 = erlang:now(),
ga:main([8,0.05,0.8],[40,0.01,0.5]),
T2 = erlang:now(),
timer:now_diff(T2,T1)/1.0e6.
```

The *f()* function is for clearing buffer. Function *now()* is the clock function available in the Erlang shell. We start the concurrent process by calling function *main()*. In the above example, concurrent genetic algorithms with two processes have been invoked. The deduction of start time from the stop time will yield the running time. We also change the time unit from microsecond to second. The concurrent genetic algorithms coding can be easily adjusted to spawn more than two processes. Running time of 2 to 128 processes have been summarized and shown in Table 1.

Table 1. Running time (in seconds) of sequential and concurrent genetic algorithms.

Number of Processes	Running time (seconds)		Time Reduction	Time Saving (%)
	Concurrent GA	Sequential GA		
2	0.613	4.49	3.877	86.34744
4	0.749	4.49	3.741	83.31849
8	0.499	4.49	3.991	88.88641
16	0.484	4.49	4.006	89.22049
32	0.592	4.49	3.898	86.81514
64	2.481	4.49	2.009	44.74388
128	5.319	4.49	-0.829	-18.4633

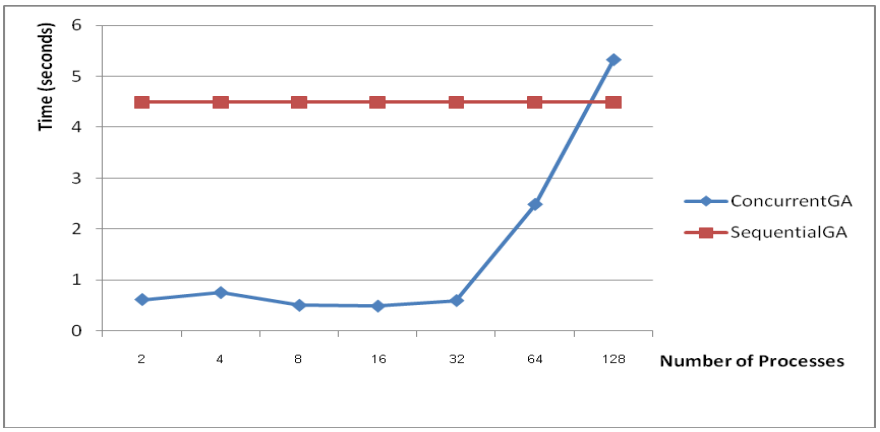

Fig. 4. Computational time comparison of sequential versus concurrent genetic algorithms.

It can be seen from the experimental results (Figure 4) that concurrent genetic algorithms with 16 processes give the best computation performance. When the number of spawned processes is higher a hundred, concurrency yields poorer performance than serial computation. This is mainly because every time the main process spawn a child process, there is an overhead cost of message passing. For this specific simple problem, we should not concurrent more than 16 processes. The optimal number of processes is however subjective and varied according to the problem domain. Empirical study is essential for the best parameter setting.

6 Conclusion

Genetic algorithms have gained interest because of their robust search characteristics. The algorithms mimic an evolutionary process of the nature such that at each generation of the search process, individuals with higher fitness values are considered good candidate solutions. The algorithms compensate the lost parts of unexplored search space by the crossover and mutation mechanisms. Genetic algorithms can therefore find the best solution quickly.

We are interested in tuning the search performance of genetic algorithms by the programming technique known as concurrency. In the paper, we illustrate concurrent genetic algorithms implemented with Erlang. The functional style of Erlang provides advantage to the minimal size of codes, whereas its concurrency facility supports the rapid implementation of concurrent processes. The source codes provide in the paper is a simple form of concurrent model. We plan to extend our research to a more effective model. Other kinds of genetic algorithms such as adaptive algorithms and neuro-genetic algorithms are also in the main line of our future research direction.

Acknowledgments. This work has been supported by grants from the National Research Council of Thailand (NRCT) and Suranaree University of Technology via the funding of Data Engineering Research Unit.

References

1. Adeli, H., Cheng, N.-T.: Concurrent genetic algorithms for optimization of large structures. J. of Aerospace Engineering 7, 3, 276--296 (1994)
2. Al-Ansary, M.D., Deiab, I.M.: Concurrent optimization of design and machining tolerances using the genetic algorithms method. Int. J. of Machine Tools and Manufacture 37, 12, 1721--1731 (1997)
3. Alba, E., Chicano, F., Ferreira, M., Gomez-Pulido, J.: Finding deadlocks in large concurrent java programs using genetic algorithms. In: 10th Int. Conf. on Genetic and Evolutionary Computation, 1735--1742 (2008)
4. Bienz, A., Fokle, K., Keller, Z., Zulkoski, E., Thede, S.: A generalized parallel genetic algorithm in Erlang, In: Midstates Conf. on Undergraduate Research in Computer Science and Mathematics (2011)
5. Cantu-Paz, E.: Markov chain models of parallel genetic algorithms. IEEE Transactions on Evolutionary Computation 4, 3, 216--226 (2000)
6. Genco, A., Lopes, S., Lo Re, G., Tartamella, M.: Routing optimization by concurrent genetic algorithms. In: 4th Int. Conf. on Applications of High-Performance Computing in Engineering, 332--339 (19950
7. Holland, J.H.: Adaptation in Natural and Artificial Systems. Univ. of Michigan Press (2004)
8. Lim, D., Ong, Y.-S., Jin, Y., Sendhoff, B., Lee, B.-S.: Efficient hierarchical parallel genetic algorithms using grid computing. Future Generation Computer Systems 23, 658--670 (2007)
9. Sehitoglu, O.T., Ucoluk, G.: Gene level concurrency in genetic algorithms. In: Int. Symp. on Computer and Information Sciences, 976--983 (2003)
10. Tagawa, K.: A statistical study of concurrent differential evolution on multi-core CPUs. In: The Italian Workshop on Artificial Life and Evolutionary Computation, 1--12 (2012)
11. Zamanifer, K., Koorangi, M.: Designing optimal binary search tree using parallel genetic algorithms. Int. J. of Computer Science and Network Security 7, 1, 138--146 (2007)