Concurrent Data Mining and Genetic Computing Implemented with Erlang Language

Kittisak Kerdprasop and Nittaya Kerdprasop

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

Abstract

The discovery process of data mining concerns an automatic extraction of interesting patterns and correlations from a large database. These patterns can reveal implicit relationships among set of objects that lead to the generation of actionable rules to be used for financial forecast, medical diagnosis, and many other useful applications. Current studies in data mining and genetic computing concentrate on how to effectively find all objects frequently co-occurring or correlated. For a massive database, parallel method is a solution for the scalability problem. In this paper, we present the design of parallel methods to the genetic algorithms, clustering, and association mining tasks. The implementation of the proposed method is based on the concurrent functional programming paradigm using the Erlang language that handles parallelism via a message passing mechanism. We test our implementations on the synthetic data sets and the real genetic data. The results show a good runtime improvement.

Keywords: Concurrent programming, Erlang language, Concurrent genetic algorithms, Concurrent clustering, Concurrent association mining

1. Introduction

Concurrent computing is a form of parallel task computation. A task-parallel method is commonly used in computer programming [12, 17, 23, 28] to speedup the computational time. Currently, there are two commonly used concurrent computing methods: thread and message passing. OpenMp is a well-known thread implementation [10], whereas MPI and MapReduce [30] are examples of message passing techniques. In this paper, we study parallelization of data mining and genetic computing tasks based on the message-passing method using Erlang language [5]. This language uses concurrent functional paradigm and communicates among hundreds of active processes via simple message passing built-in functions. As an example, to create multiple processes in Erlang, we use a spawn function as follows:

-module(example).
-export([start/0]).

start() ->
    Pid1 = spawn(fun run/0),
    io:format("New process ~p~n", [Pid1]),
    Pid2 = spawn(fun run/0),
    io:format("New process ~p~n", [Pid2]).

run() -> io:format("Hello ! ~n", []).
The start function in a module example, which is the main process, creates two processes with identifiers Pid1 and Pid2, respectively. The newly created processes execute a `run` function that prints the word “Hello!” on the screen as the following:

```
New process <0.63.0>
Hello!
New process <0.64.0>
Hello!
```

The numbers <0.63.0> and <0.64.0> are identifiers of the newly created two processes. Each process then independently invokes the `run` function to print out a word “Hello!” on the screen. The processes in Erlang virtual machine are lightweight and do not share memory with other processes. Therefore, it is an ideal language to implement large scale parallelizing algorithms through the concurrent computation methods.

2. Concurrent implementation methods

2.1. Concurrent genetic algorithms

Genetic algorithms are search and optimization methods inspired by the natural selection process that causes biological evolution [11]. 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 [1, 3, 4]. Parallel computation for genetic algorithms has been proposed [7, 18, 24, 25] for at least two decades to speedup the computational time. Our work presented in this paper propose a simple 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., [6] is close to ours, but their process interaction scheme is more tightly coupled than our scheme.

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 population 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(_, 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").

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.

Figure 1. A message-passing model in concurrent genetic algorithms
module(ga).
-compile(export_all).
main([P1,M1,C1],[P2,M2,C2]) ->
    Pid2 = spawn(MODULE, process, [P1,M1,C1]),
    Pid3 = spawn(MODULE, process, [P2,M2,C2]),
    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.

Figure 2. Implementation and running result of concurrent genetic algorithms with two active 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.

2.2. Concurrent clustering

Clustering is an unsupervised learning problem widely studied in many research areas such as statistics, machine learning, data mining, pattern recognition. The objective of clustering process is to partition a mixture of large dataset into smaller groups with a general criterion that data in the same group should be more similar or closer to each other than those in different groups. A serial k-means algorithm was proposed by J.B. MacQueen in 1967 [20] and since then it has gained much interest from data analysts. Despite its simplicity and great success, the k-means method is known to degrade when the dataset grows larger in terms of number of objects and dimensions [13, 15]. To obtain acceptable computational speed on huge datasets, most researchers turn to parallelizing scheme [9, 14, 16, 22, 27, 30].

The serial k-means algorithm [20] starts with the initialization phase of randomly selecting temporary \( k \) central points, or centroids. Then, iteratively assign data to the nearest cluster and then re-calculate the new central points of \( k \) clusters. The serial algorithm takes much computational time on calculating distances between each of \( N \) data points and the current \( K \) centroids. Then iteratively assign each data point to the closest cluster. We thus improve the computational efficiency by assigning \( P \) processes to handle the clustering task on a smaller group of \( N/P \) data points. The centroid update is responsible by the master process. In Figure 3, the PKM algorithm is the master process responsible for creating new parallel processes, sending centroids to the created processes, receiving the cluster assignment results, and recalculating the new centroids. The steps repeat as long as the old and the new centroids do not converge. The convergence criterion can be set through the function \( \text{difference}(C, C') \). A screenshot of compiling and running the program (Erlang release R13B04) is given in Figure 4.
Figure 3. Communication between master and created processes in the concurrent k-means clustering

![Diagram of communication between master and processes]

- module(pkm).
  - import(lists, [seq/2, sum/1, flatten/1, split/2, nth/2]).
  - import(io, [format/1, format/2]).
  - import(random, [uniform/1]).

start(DataL, Cent, NumPar) ->
  CidL = myspawn(NumPar),
  LastC = myloop(CidL, Cent, DataL, NumPar),
  format("~w", [LastC], LastC).

myspawn(0) -> [];         
myspawn(N) -> [spawn(?MODULE,c,[self()])|myspawn(N-1)].

myloop(CidL, Cent, DataL, NumPar, Count) ->
  mysend(Count, CidL, Cent, DataL),
  L = flatten(myrec(Count, NumPar)),
  C = calNewCent(Cent, L),
  format("~w.", [Count]),
  if Count >100 -> mystop(CidL), C;
  Cent/= C_ ->
    myloop(CidL, C_, DataL, NumPar, Count+1);  
  true -> mystop(CidL), C_
end.
c(Sid) ->
  receive
    stop -> true;
    {LoopN, Cent, Data} ->
      L = locate(Data, Cent),
      Sid!{LoopN, L}, c(Sid)
  end.

Figure 4. Coding and series of line commands to create four initial centroids (command 3), then partition 800,000 data points into eight subgroups sending to the eight processors (command 4), parallel k-means (pkm) starts at command 5. The outputs of pkm are the number of iteration (i.e., 35) and the mean points of four clusters. A variable TReal is for displaying the running time of the whole process, which is 129371000 microseconds or 129.371 seconds, including send-receive messages between the master and the eight concurrent processes.
2.3. Concurrent association mining applied to the splice site recognition problem

The splice site recognition problem can be formulated as the following. Given some part of unclassified genomic DNA sequences, decide whether this is an intron/exon border, an exon/intron border, or none of the two splice sites. To develop an accurate prediction model, a machine learning technique is usually applied. The learning task is that given sequences of genomic DNA with known splice junction labeled as either an intron/exon, an exon/intron, or none, the learning objective is to find a classification rule that can successfully predict the region of uncharacterized genomic DNA sequence.

Splice site prediction can be considered as a subproblem of gene prediction that aims at correctly recognizing gene from the given fragment of DNA sequence. The task of splice site prediction is to recognize the actual boundaries of the protein-coding regions in the DNA sequence. There are many computational techniques applied to tackle this problem. The direct method [8, 29] is to analyze the sequence to capture gene profile and identify specific features that can accurately predict the splice junctions. Researchers from the machine learning community prefer to attack this problem via a single or multiple classification learning algorithms [19, 21].

Our approach to solve the splice site recognition problem is different from those appeared in the literature in that our predictor is built from the association analysis technique [2], not the classification ones. The advantage of the proposed technique is that the prediction model can contain nucleotides at arbitrary position, not necessarily be the contiguous base sequences.

At the initial stage of our proposed method (named assoDNA), the training dataset with a mixture of exon/intron, intron/exon, and none of the two DNA sequence splice sites is separated into three subsets according to splice junction types. Each data subset is then processed through the same steps of frequent patterns and association analysis. Once the three data subsets are processed through the frequent pattern analysis method, the three sets of learning results (displayed as prediction rules) are finally combined and prioritized according to the confidence and support values. The proposed assoDNA method can be explained as a flow diagram shown in Figure 5.

The concurrent implementation of assoDNA is illustrated as follows:

```prolog
-module(assoDNA_par).

concurrent(P1, P2, P3) ->
    spawn(assoDNA_par, run, [self(),P1]),
    spawn(assoDNA_par, run, [self(),P2]),
    spawn(assoDNA_par,run,[self(),P3]),
    receive
        my_end -> ok
    end.

run(MasterID, InputL) ->
    R = main2(any, 3, InputL),
    file:delete("out.txt") ,
    AD = lists:last(R), [ADD|_] = AD,
    Rules = lists:sublist(R, length(R)-1),
    PrintRules = map(fun({D, S, Per, Class}) ->
        {to_Col3(notLast(D)),S,Per,transformBack(Class)} end,
        Rules),
    ADP=lists:map(fun(Data) -> {Data,checkRules(Data,Rules)} end, AD),
    ADPprint=map(fun({Data, V}) ->
```

The concurrent implementation of assoDNA is illustrated as follows:
Predict = transformBack(V),
{Data, [last(Data), Predict,
mark(last(Data), Predict)]} end, ADP),
Predict = map(fun({F,S}) -> {to_Col3(notLast(F)),S} end, ADPprint),
writeToFile(Predict),
[_ ,Stop|_] = InputL,
if Stop == 2 -> MasterID ! my_end ;
true -> MasterID ! not_end
end.

Figure 5. A flow diagram illustrating the assoDNA method
3. Performance Study Results

3.1. Performance of concurrent genetic algorithms

We design a series of experimentation to compare performance of sequential genetic algorithms against the concurrent implementation. 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:

```erlang
defined($f$),
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 $\text{now}()$ is the clock function available in the Erlang shell. We start the concurrent process by calling function $\text{main}()$. 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 graphically shown in Figure 6. It can be seen from that concurrent genetic algorithms with 16 processes give the best computation performance. When the number of spawned processes is higher than 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.

![Figure 6. Computational time comparison of sequential versus concurrent genetic algorithms](image)

---

Image: Example of a figure from the document.

---

71
3.2. Performance of concurrent clustering

We evaluate performances of the proposed PKM algorithm on synthetic two dimensional dataset. The computational speed of concurrent k-means as compared to serial k-means is given in Figure 7. It is noticeable that when dataset is small (N=50), running time of concurrent k-means is a little bit longer than the serial k-means. This is due to the overhead of spawning concurrent processes. At data size of 900,000 points, running time is unobservable because the machine is out of memory. Speedup advantage is very high (more than 30%) at dataset of size between 50,000 to 200,000 points.

Figure 7. Running time comparisons of serial versus parallel k-means

3.3. Performance of concurrent association mining

The dataset used in this work is primate splice-junction gene sequences available at the UCI repository of machine learning databases [26]. This dataset are taken from GenBank 64.1 containing 3,190 DNA sequences. Each sequence is a window of 60 DNA base pairs starting at position -30 and ending at position +30 corresponding to the splice site location. The splice junction can be either a junction between intron and exon (intron/exon), a junction between exon and intron (exon/intron), or no junction at all (none).

To improve the computational performance of the proposed assoDNA method, we employ the concept of concurrent programming. Reduction in running time can be compared through the screenshots in Fig. 8 in which the last line on a upper screen is sequential running time (in a unit of microseconds), whereas the last line on a lower screen is concurrent running time. Time reduction in this example is around 46.29%.
4. Conclusion

Data mining and soft computing via genetic algorithms share a common goal of extracting patterns and useful information from a large collection of data. One important problem of such intelligent techniques is scalability due to huge amount of data to be processed. In this paper, we propose the design and implementation of concurrent computation to speedup the execution time over large amount of data. We investigate the robust search technique of genetic algorithms and propose that the algorithms can be improved via concurrency. The data mining tasks presented in this paper are clustering and association mining. Their performances also improved significantly with the concurrency scheme. From these promising results, we plan to further our study over distributed concurrent method.

Acknowledgements

This research was supported by the SUT Research and Development Fund, Suranaree University of Technology.
References


Authors

Kittisak Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology, Thailand. He received his bachelor degree in Mathematics from Srinakarinwirot University, Thailand, in 1986, master degree in computer science from the Prince of Songkla University, Thailand, in 1991 and doctoral degree in computer science from Nova Southeastern University, USA., in 1999. His current research includes Data mining, Artificial Intelligence, Functional and Logic Programming Languages, Computational Statistics.

Nittaya Kerdprasop is an associate professor at the school of computer engineering, Suranaree University of Technology, Thailand. She received her B.S. from Mahidol University, Thailand, in 1985, M.S. in computer science from the Prince of Songkla University, Thailand, in 1991 and Ph.D. in computer science from Nova Southeastern University, USA, in 1999. She is a member of ACM and IEEE Computer Society. Her research of interest includes Knowledge Discovery in Databases, Artificial Intelligence, Logic Programming, Deductive and Active Databases.