Gene Level Concurrency in Genetic Algorithms

Onur Tolga Şehitoğlu and Göktürk Üçoluk

Computer Engineering Department, METU, Ankara, Turkey,
{onur,ucoluk}@ceng.metu.edu.tr

Abstract. This study describes an alternative concurrency approach in genetic algorithms. Inspiring from implicit parallelism in a physical chromosome, a vertical concurrency is introduced. Proposed gene process model allows genetic algorithms work in encodings independent from the gene position ordering in a chromosome. This feature is used to implement a gene reordering version of genetic algorithm. Further possible models of flexible gene position encodings are discussed.

1 Introduction

Genetic algorithms mimic the evolution in the nature in order to search a solution space for non-trivial problems like optimization problems. They have been successfully applied to many engineering and scientific problems where a polynomial time deterministic algorithm does not exist. They efficiently explore of the search space by using the global information gained during the exploration in a population of individuals.

As genetic algorithms get widely used, parallel genetic algorithms became a major study area. Cantú-Paz classifies parallel genetic algorithms in three groups [1]: In *global parallelism* application of genetic operators and evaluation of individuals are explicitly parallelized. In *coarse grain* and *fine grain* parallelism population is distributed among the processors and only mating within the subpopulations and among the neighbor populations are allowed. Also there are some *hybrid* studies combining these approaches.

In this study a different approach to fine grain parallelism is discussed where each process keeps track of a single gene value of the whole population. Crossover and mutation operations can be done concurrently as single cycle operations providing an implicit parallelism for genetic operators. In addition, operations become independent from the gene ordering making more flexible chromosome encodings possible [2,3].

In the following sections, first the gene process model where each process represents a gene position is introduced. Then application of this model with a gene reordering genetic algorithm [4,3] is described. Then other possible encodings in this model are discussed.

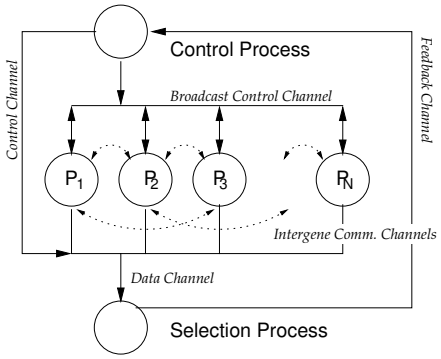

Fig. 1. Architecture of the gene process model

2 Gene Process Model

In the proposed model, each gene position in the phenotype is executed by a separate process. In each process an object keeps the specific genom data of the whole population. These objects are connected to each other and to two other special processes via communication channels. These two special processes are the *Control Process* and the *Selection Process*. Architecture of the model is given in Figure 1.

Execution of the entire model is based on message passing on order preserving and lossless channels. All gene processes and the selection process execute a *receive message — do corresponding action* loop. The control process initiates and supervises the genetic algorithm.

If we represent the i^{th} chromosome in the population as a vector of genes as $\Gamma_i = \langle g_1, g_2, \ldots, g_n \rangle$ where n is the phenotype length, and the population size is m, we can denote the whole population $\mathbf{\Gamma}$ as a vector of chromosomes. A particular gene value in the population is then represented as $\Gamma_{i,j}$ denoting the j^{th} gene of the i^{th} chromosome.

In the proposed architecture, this representation is transposed into a collection of vertical gene vectors instead of a collection of chromosomes. A gene vector Λ_i is the vector containing all gene values in the i^{th} position of the chromosomes of the pool $\Lambda_i = \langle \Gamma_{1,i}, \Gamma_{2,i}, \ldots, \Gamma_{m,i} \rangle$.

All gene processes are identical objects defined by a set of attributes and actions (methods). So a gene process P_i is a tuple:
$P_i = \langle i, \mathsf{pos}, \Lambda, new\Lambda, \mathsf{Broad}, \mathsf{Inport}, \overline{\mathsf{Outports}}, \mathsf{Actions} \rangle$
Where:

- i: Position of the gene implemented by this process in the phenotype.
- pos: Phenotype independent position information. It is used by variable permutation representations and may contain any spatial or hierarchical information. In simple GA implementation it is set to i.
- Λ: Gene vector Λ_i of the current population.

$new\Lambda$: Gene vector Λ_i of the new generation.
Broad: Broadcast command channel to be listened
Inport: Private input channel of the process
Outports: A vector of output ports that this process uses to send data. It includes the data channel flowing through the *selection process*, the inter-gene channels and the broadcast channel. In simple GA implementations only the data channel is sufficient.
Actions: A set of actions. Each action is identified by a named tuple of any arity. When a message of the same pattern is received the corresponding action is executed by the gene process.

A gene process executes a very simple command sequence:
1) Wait for a message in Inport or Broad. 2) Execute the matching action. 3) Go to step 1

Selection process S is similarly a tuple:
$S = \langle \mathbf{\Gamma}, new\mathbf{\Gamma}, \overline{\mathsf{fitnesses}}, \mathsf{Inport}, \mathsf{Outport}, f, \mathsf{Actions} \rangle$
Where:
$\mathbf{\Gamma}$: Data of the entire pool of the current generation.
$new\mathbf{\Gamma}$: Data of the entire pool of the new generation.
fitnesses: Fitnesses calculated for the individuals in the pool
Inport: Input channel to be listened.
Outport: Feedback channel to send results of the selection.
f: Fitness function.
Actions: A set of actions. Similar to gene processes.

Selection process executes the same loop with the gene processes:
The pool data is received from the gene processes. Then the fitness calculation and the selection action is initiated by a message coming from the control process. The selection information is sent back to the control process via the feedback channel.

The control process executes the genetic algorithm loop by sending genetic operator messages to all gene processes, than the fitness calculation message and the selection message to the selection process, in each iteration. Repeating this iteration for many generations will make genetic algorithm work.

3 Simple GA Implementation

In a simple GA implementation a gene process implements the following messages:

`init(`M`)` Initializes the gene vector Λ randomly

$$\Lambda_i \leftarrow random() \text{ for all } i = 1, .., M$$

`mutate(`*prob*`)` All genes in the new population $new\Lambda$ is mutated with a probability of *prob*.

$$new\Lambda_i(t{+}1) \leftarrow \begin{cases} mutated(new\Lambda_i(t)) & \text{if } random([0,1]) \leq prob \\ new\Lambda_i(t) & \text{otherwise} \end{cases} \text{ for all } i = 1, .., M$$

`crossover(`cp`, `$p1$`, `$p2$`)` Parents $p1$ and $p2$ in the current population are crossovered to produce two new offsprings in the new population. cp is the crossover point where offsprings switch parents. ($\bullet$ is used to denote the append operation)

$$new\Lambda \leftarrow new\Lambda \bullet \langle g_A, g_B \rangle \text{ where } \langle g_A, g_B \rangle = \begin{cases} \langle \Lambda_{p1}, \Lambda_{p2} \rangle & \text{if } \mathsf{pos} \leq cp \\ \langle \Lambda_{p2}, \Lambda_{p1} \rangle & \text{otherwise} \end{cases}$$

`uniformcrossover(`$p1$`, `$p2$`,[`$prob$`])` In uniform crossover, offspring value is taken from one of the parents, probabilistically.

$$new\Lambda \leftarrow new\Lambda \bullet \langle g_A, g_B \rangle \text{ where } \langle g_A, g_B \rangle = \begin{cases} \langle \Lambda_{p1}, \Lambda_{p2} \rangle \text{ if } random([0,1]) \leq prob \\ \langle \Lambda_{p2}, \Lambda_{p1} \rangle \text{ otherwise} \end{cases}$$

`senddata()` This messages initiates the sending of the gene information to the selection process so that it could make calculations on this global data. $\mathsf{send}_{datachannel}$`(data(`$i$`, `$new\Lambda$`))`

`select(`$List$`)` The control process sends a list of chromosome identifiers to be selected via this message.

$$\Lambda \leftarrow \langle new\Lambda_k \mid k \in List \rangle \quad ; \quad new\Lambda \leftarrow \langle\, \rangle$$

Similar to the gene processes, selection process implements a set of messages as actions. Those messages are received over the same channel from gene processes or control process. This messages are `data(`i`,`L`)`, for getting the data of the i^{th} gene process, `setfitnesses()` for calculating fitnesses over the population, `select(`K`)` to select K element for crossover in the next generation by applying any selection mechanism, `sendselected()` to send the list of selected individuals to the control process back.

After this definitions of messages (or actions) control process executes the following loop:

send_$broadcast$`(init(`M`))`
`DO {` `REPEAT` ($CrossoverRatio \times M$) times {
$A,\ B \leftarrow random(1,M)$; $CP \leftarrow random(1,N)$
send_$broadcast$`(crossover(`CP, A, B`))` }
send_$broadcast$`(mutate(`$MutateProb$`),senddata())`
send_$selection$`(setfitnesses(),select(`M`),sendselected(`M`))`
receive_$feedback$`(selected(`$List$`))`
send_$broadcast$`(select(`$List$`))`
} `WHILE` Maximum number of generations is not reached

Most of this communication can be done asynchronously. Only places requiring synchronization are the fitness calculation and the selection process.

4 Gene Reordering GA

Gene reordering genetic algorithm is introduced in [4,3]. It is a linkage learning method and empirically shown to improve online performance of genetic algorithms in deceptive problems.

In gene reordering GA, a single global permutation is considered during the computation. This permutation maps a gene number to a position value in the chromosome encoding. Crossover operations are based on this mapping instead of the original gene ordering in the representation. During the evolution of the genetic algorithm, this global permutation is readjusted by means of statistical analysis on neighboring genes in the population.

Let $\mathbf{\Gamma}$ be the encoding of the problem and Γ_i represent the value of the i^{th} feature in a chromosome. We define a permutation function $p : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$ so that Γ_i will be considered in the $p(i)^{th}$ position for crossover operation and $\Gamma_{p^{-1}(i)}$ and $\Gamma_{p^{-1}(i+1)}$ values are neighbor gene values with respect to p.

We define a *neighbour-affinity function* for neighbour gene positions in the permutation in its most general form as: $\mathcal{A}_i^p : \{\langle \Gamma_{p^{-1}(i)}, \Gamma_{p^{-1}(i+1)} \rangle\}_{pool} \mapsto [0, 1]$

$\mathcal{A}_i^p$ function is based on a statistical analysis of the pool and maps the neighbour values from the pool to an affinity value. The type of the statistical analysis used depends on the problem representation. Examples can be *hamming distance* for binary encoding and *variance* for other numerical encodings.

The next step is to modify the permutation mapping by looking at the results of neighbour-affinity value calculations from the individuals in the current population. We will be calling these values A_i^p. All A_i^p values will be calculated with some period of generation. If an A_i^p value is found to be less then a threshold value τ, these neighbour genes will be considered not to contribute solution well together and the permutation will be changed to separate them. Actually there exist 4 possible affinity cases for each gene position:

1. *Affinity is greater then τ for both neighbours* so current ordering is fine. The gene position in the permutation kept unaltered.
2. *Affinity with left gene is greater than τ but it is less than τ for the right gene*. So left neighbour is fine but we should separate it from the right. So it exchanges position with the left neighbour.
3. *Affinity with left gene is less than τ but it is greater than τ for the right gene*. Similarly it exchanges position with the right neighbour.
4. *Both affinities are less than τ*. Gene is moved to a random position in the permutation.

In this way, modifications in the permutation mappings are kept to be as local as possible.

5 Gene Reordering Genetic Algorithms and Gene Level Concurrency

In *gene reordering genetic algorithms* global ordering of genes with respect to crossover operator is changed by the local decisions to maximize affinity values among the gene positions.

In gene level concurrency, crossover operation is based on the pos attribute of each gene process. So moving a gene in the global permutation is just simply

updating the pos values of the gene processes. Since the affinity calculations are done among the neighboring genes, they can be calculated by means of local communications. In this way simple genetic algorithm implementation can be extended into gene reordering genetic algorithm with a few additions.

In the gene reordering genetic algorithm, in addition to simple GA, a gene process includes LeftAffinity and RigthAffinity attributes. Furthermore, $\overline{\mathsf{Outports}}$ includes private ports to the other gene processes.

The new gene process is the following tuple:
$P_i = \langle i, \mathsf{pos}, \Lambda, new\Lambda, \mathsf{Broad}, \mathsf{Inport}, \overline{\mathsf{Outports}}, f, aff, \mathsf{Actions}, \mathsf{LeftAff}, \mathsf{RightAff} \rangle$
Only affinity values to the left and right neighbor processes and the user defined affinity function aff are added. This *neighborhood* is defined as to possess consecutive pos attribute values.

New message and actions are required to calculate and process affinities. Affinity calculation requires the data of the two processes. So each process sends its data to the left neighbor. Left neighbor calculate and set the right affinity and sends it to the right so that it can update the left affinity. When this cycle is completed all affinity values would be known for all gene processes.

The next step is to update the positions based on the affinities obtained. Since the position updates are local in the implementation of the gene reordering genetic algorithm, this decision is left to the gene process. In the sequential implementation this is done deterministically in a linear scan of the permutation. In the concurrent version this is done probabilistically.

Following messages are introduced in order to implement gene reordering GA:

`sendtoleft()` This initiates the gene vector exchange among the neighbors. Each gene process sends its Λ to the left neighbor (having pos-1 position) in a `data(...)` message.

$$\mathsf{send}_{gene(\mathsf{pos}-1)}(\mathtt{data}(i, \Lambda))$$

`data(`k`,`$Neigh\Lambda$`)` Data is available from the right neighbor (having pos+1 position). The gene calculates and updates the right affinity by calling a user defined aff function and sends back the calculated value to its neighbor.

$$\begin{aligned} &\mathsf{RightAff} \leftarrow aff(\Lambda, N\Lambda) \\ &\mathsf{send}_{gene(\mathsf{pos}+1)}(\mathtt{setaff}(i, \mathsf{RightAff})) \end{aligned}$$

`setaff(`k`,`v`)` Left neighbor is sending the calculated affinity. Just set the LeftAff to the value sent.

$$\mathsf{LeftAff} \leftarrow v$$

`moveme(`k`,currpos,topos)` This messages is received by all processes via the broadcast channel since order change of a process may affect positions of all genes. After this change the gene process listens to the channel of its new

position. The gene requesting the move, simply updates its position.

$$\mathsf{pos} \leftarrow \begin{cases} \mathsf{topos} & \text{if } \mathsf{currpos} = \mathsf{pos} \\ \mathsf{pos}-1 & \text{if } \mathsf{currpos} < \mathsf{pos} \text{ and } \mathsf{topos} > \mathsf{pos} \\ \mathsf{pos}+1 & \text{if } \mathsf{currpos} > \mathsf{pos} \text{ and } \mathsf{topos} < \mathsf{pos} \\ \mathsf{pos} & \text{otherwise} \end{cases}$$

Only `sendtoleft(...)` is initiated by the control process. `data(...)` and `setaff(...)` is generated among the gene processes as a consequence of this message. `moveme(...)` on the other hand is a probabilistically generated message. After a gene process completes a `select(...)` request, it probabilistically (with a probability of $\frac{1}{t}$ for an expected period of t (generations) makes a movement decision and sends a `moveme(...)` message to the broadcast channel.

This decision is based on its left and right neighbor affinities and a threshold value τ, of them. If gene process have good affinities with both neighbors it will not send any message and retain its existing position. If any of the neighbour affinity is below the threshold, it sends a command to move itself into a local or a random position according to the decision specified in Section 4.

After introducing these messages into the gene processes, the only additional update required for the control process is to have a `sendtoleft(...)` message to initiate the affinity calculation in the main loop.

6 Conclusion and Other Possible Architectures

Gene reordering GA is not the only possible architecture that can be implemented in the gene level concurrency model. Since the model eliminates the requirement for a linear fixed order, any non-linear and/or variable topology can be implemented by using it.

For example a matrix notation can be suitable to represent 2-D spatial chromosome encodings (Figure 2(b)). Image processing is a good example for such an application area. The only change required is to have a tuple for the pos attribute. The crossover point is going to be any shape in the 2-D representation. This shape will separate the chromosome into two partitions. So the crossover will decide about a particular gene of a chromosome, considering which partition it belongs to.

So resulting gene process tuple will be:

$$P_i = \langle\langle i_x, i_y\rangle, \langle \mathsf{pos}_x, \mathsf{pos}_y\rangle, \Lambda, new\Lambda, \mathsf{Broad}, \mathsf{Inport}, \overline{\mathsf{Outports}}, \mathsf{Actions}\rangle$$

Any shape description which divides the matrix into two partitions is suitable. It can be a line or any arbitrary shape as long as a function returning a binary choice $partition : Shape \times Position \rightarrow \{0, 1\}$ can be defined. Than the crossover can be based on this function.

For example if the shape is a line passing from two points $\langle a_x, a_y\rangle$ and $\langle b_x, b_y\rangle$ the *partition* function can be defined as:

$$partition(\langle a_x, a_y\rangle, \langle \mathsf{pos}_x, \mathsf{pos}_y\rangle) = \begin{cases} \text{if } \mathsf{pos}_y < \dfrac{b_y - a_y}{b_x - a_x}(\mathsf{pos}_x - a_x) + a_y\ , & 0 \\ \text{otherwise}\ , & 1 \end{cases}$$

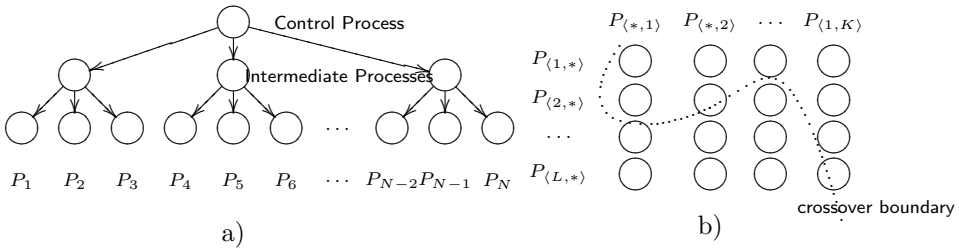

Fig. 2. Possible chromosome representations a) hierarchical representation. b) mesh representation

Another possible extension of the method is the hierarchical chromosome representation shown in (Figure 2(a)). Instead of a single level orientation of gene processes an intermediate level of processes can be inserted between the control process and the gene processes. These processes capture the incoming control messages and generate modified messages through their child processes. This allows different levels of genetic operator and building block interactions possible similar to [5].

In summary, this study describes an alternative way of realizing the implicit concurrency within a physiological chromosome and considering each genotype a separate processing identity. The resulting implementation has an implicit parallelism. This parallelism has been combined with the encoding flexibility of the proposed model. Gene reordering genetic algorithm is an example of this flexibility work. This idea of having each gene position a separate entity can be further expanded to have alternative approaches to genetic algorithm.

References

1. Cantú-Paz, E.: A summary of research on parallel genetic algorithms. Technical Report 95007, IlliGAL (1995)
2. Sehitoglu, O.T.: A concurrent constraint programming approach to genetic algorithms. In Ryan, C., ed.: Graduate Student Workshop, San Francisco (2001) 445–448
3. Şehitoğlu, O.T.: Gene Reordering and Concurrency in Genetic Algorithms. PhD thesis, Computer Engineering Dept, METU, Ankara (2002)
4. Sehitoglu, O.T., Ucoluk, G.: A building block favoring reordering method for gene positions in genetic algorithms. In Spector, L., Goodman, E., eds.: Proc. of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, Morgan Kaufmann (2001) 571–575
5. Pelikan, M., Goldberg, D.E.: Escaping hierarchical traps with competent genetic algorithms. In Spector, L., Goodman, E., eds.: Proc. of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, Morgan Kaufmann (2001) 511–518
6. Holland, J.: Adaptation in Natural and Artificial Systems. MIT Press (1975)
7. Harik, G.: Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms. PhD thesis, University of Michigan (1997)