A Unified Paradigm for Parallel Genetic Algorithms

A. Kapsalis, G.D. Smith and V.J. Rayward-Smith

University of East Anglia
e-mail: ak@sys.uea.ac.uk

Abstract

Genetic Algorithms contain natural parallelism. There are two main approaches in parallelising GAs. The first one is parallelising individual functional components of a standard, sequential GA. The only difference with the sequential GA is in the computation speed. The second approach more closely resembles the real life simultaneous evolution of species, which is the central theme in GAs. Algorithms following this approach are still referred to as GAs but are different from Holland's standard GA. For these algorithms it is not the improvement in computation speed that is the driving factor, but the efficiency with which they search a given solution space. We describe a number of the most common parallel GA methods found in the literature and mention practical issues concerning their implementation in a Transputer based system. We go on to introduce the Unified Parallel GA system, based on our GA toolkit, GAmeter, which allows the user to select one or more of the GA methods described, by setting various parameters. Finally we present results for the Steiner tree Problem in Graphs (SPG).

1. Introduction

The term Genetic algorithms (GAs) refers to a family of computational models inspired by Darwin's theory of evolution and based on principles derived from natural population genetics. The natural processes of reproduction, selection and survival of the fittest are simulated and used to give good approximate solutions to a wide range of optimisation problems. The foundations of the technique were described by Holland [Holl75], who also established much of the theory to explain the subsequent success of the application of GAs to a wide variety of problems.

Research on improving the performance of Holland's "canonical" GA has created a vast number of alternative algorithms based on the same principles. The algorithm for the standard/sequential GA (SGA), shown in figure 1, is a generalisation that covers most of the various instantiations of the sequential GAs. In fact, it was reported in [Kaps93a] and [Rayw94] that this algorithm can be used to describe other search methods such as hill climbing, simulated annealing and tabu search.

The GA starts by creating an initial (usually random) pool, \( P \), of chromosomes or "genetically" described solutions to a problem. These chromosomes are \textit{Evaluated} and a fitness score is associated with them. At each iteration a number of Parent chromosomes, \( Q \), are \textit{Selected} from the pool based on their fitness scores. The selected chromosomes \textit{combine} using genetic operators like \textit{crossover} and \textit{mutation} to \textit{Create} the set \( R \) of new chromosomes. The new chromosomes are evaluated and a call to function \textit{merge} determines which strings will form the pool \( P \) for the next iteration. Iterations are usually referred to as generations.
topology for such an approach is the star topology. Because there are different demands in communication depending on the degree of parallelisation used we separate our investigation of the Parallel GA (PGA) to Level1 and Level2.

**Level1**

In Level1 only the Evaluate function is parallelised. The master is a GA that transmits all chromosomes that need to be evaluated to the available slaves. The slaves wait until they receive a chromosome at which point they evaluate it and return its fitness value to the master.

**ParallelEvaluate()**

```plaintext
Initialise Queue
Send (Chromosome, ID Number) to slave at the head of the queue
While not all fitness values collected do
    begin
        if (Send completed) & (queue not empty) & (more Evaluations required) then
            Send (Chromosome, ID Number) to slave at the head of the queue;
        if there exists message to receive then
            Receive (fitness value, ID Number);
    end;

Slave

While message>0 do
    begin
        Wait to Receive (Chromosome, message);
        if message>0 then Evaluate (Chromosome,fitness value);
        Send (fitness value, message);
    end;
```

If |R| is the number of chromosomes that need to be evaluated at each generation then the total size of messages exchanged between the master and the slaves is:

\[ |R| \times (\text{StringSize} + \text{MessageSize}) \] sent by the master and
\[ |R| \times (\text{fitnessValueSize} + \text{MessageSize}) \] returned by each slave.

*Message* is an integer used as a tag or identification number so that the fitness value returned by the slave is associated with the appropriate chromosome. This is needed because the slaves might not respond in the same sequence with which they receive chromosomes. Negative valued messages are used to control the operation of the slave. Depending on the *Message* value it can wait to receive new problem data or stop operating.

**Level2**

In Level2 both Create and Evaluate are parallelised. The most common genetic operator used in create is crossover. Given two chromosomes this operator combines them and creates two new chromosomes. Every new chromosome created must be evaluated and therefore Evaluate can be seen as a component of Create. The