SIGEVOLution 13(3):2-4. 10.1145/3430913.3430914

# Multi-threaded Memory Efficient Crossover in C++ for Generational Genetic Programming

W. B. Langdon

**Abstract**

C++ code snippets from a multi-core parallel memory-efficient crossover for genetic programming are given by https://arxiv.org/abs/2009.10460 and on line at http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/efficient_memory/. They may be adapted for separate generation evolutionary algorithms where large chromosomes or small RAM require no more than $M + 2 \times$ nthreads simultaneously active individuals. We summarise arXiv 2009.10460.

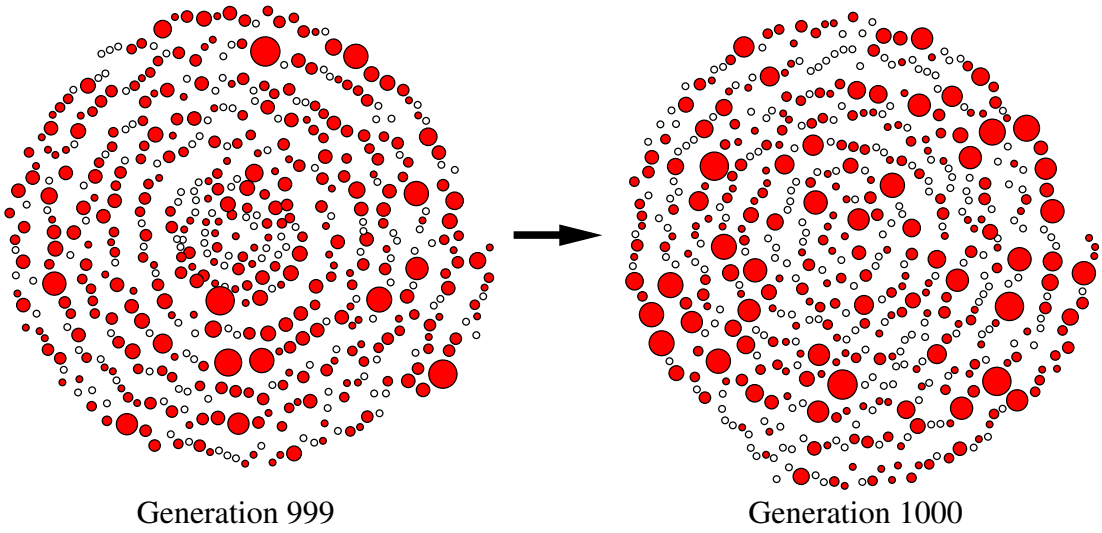


Figure 1: In generational schemes (e.g. $\mu, \mu$) the current population (on the left) is completely replaced by the next population (on the right). The area of red dots is proportional to the number of children it will be a parent of in the next generation. Small white dots are infertile. We describe how to avoid simultaneously storing both populations.

## 1 Generational can be as Memory Efficient as Steady State

It has been known for a long time that in conventional genetic algorithms using two parent crossover only $M+2$ individuals need be simultaneously stored for non-overlapping generations of $M$ individuals. Indeed this is true of generational Evolutionary Computation (EC) in general. (Steady state GAs, GPs and $(\mu+\lambda)$-ES Evolution Strategies use overlapping generations, Figure 2.) Nonetheless the widespread availability of large RAM computers and compact coding of individual chromosomes seems to have led to the $M+2$ limit being forgotten and the widespread use of inefficient computer implementations with separate new and old populations, each requiring storage for $M$ individuals. Recent work with enormous trees or large populations has prompted renewed interest in ensuring that generational implementations are as efficient as steady state implementations. (For example, see email discussion on GeneticProgramming@groups.io 5–7 September 2020.)

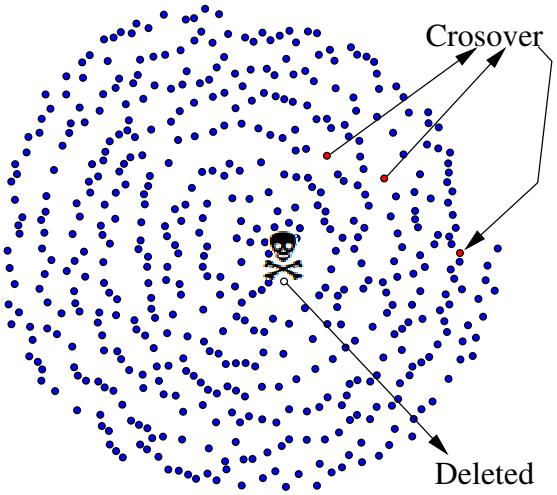


Figure 2: In steady state Evolutionary Algorithms (e.g. $\mu + 1$) parents are selected from the current population, their offspring created and inserted into the current population (red). To keep the population size constant, the same number of individuals (white) are removed from the population. Many strategies are available for selecting parents and for selecting who to remove from the population.

## 2 Minimal Memory Generational EC Algorithms

In their 1999 book (GP3) Koza et al. (pages 1044-1045) divide the current population into four classes, according to how many children they have: 0, 1, 2, more than 2. The new population is created in this priority order (0, 1, 2, 2+). Our algorithm is slightly simpler. Since Andy Singleton's GPquick uses crossover which creates one child (rather than two), this allows the "2" and "more than 2" classes to be combined into a class "two or more" (2+). We then follow GP3 except (to allow crossovers in parallel) we allocate up to two free slots per thread (rather than an extra two).

At the start of the new generation, the parent population are assigned into classes 0, 1 and 2+. All the parents without children (class 0) are deleted. The children to be created are assigned into class 1 or 2+, according to the minimum class of their (two) parents. I.e. if either parent is in class 1, the new child is placed in class 1, otherwise class 2+. With rapid parallel fitness evaluation, the cost of crossover can be significant, therefore both crossover and fitness evaluation are done by parallel threads. Note the operations in the next paragraph are also done in parallel.

Creation and testing of new individuals then starts with the class 1 individuals. Each time a new individual has been created, it is removed from the data structures for both its parents. If it has a class 1 parent, removing it will mean that that parent now no longer has any children to be processed and so is deleted. Removing it from a class 2+ parent, may mean the parent still has two or more children to be processed, so it stays in class 2+ or it may now only have one child to be created, in which case the child is moved to class 1. Class 1 children (i.e. with one or more class 1 parent) are created before those with two class 2+ parents.

Due to variations in thread timings, there can be small variations in memory usage between otherwise identical runs. To be efficient our implementation assumes: 1) run time is dominated by crossover and fitness evaluation (excluding gathering debug/performance statistics, they can be 99.9% of the cost of large GP experiments), 2) the number of cores is small compared to the population size ($M$) and 3) individual evaluation times are similar. Indeed in practice on long runs it is efficient even with very different tree evaluation times, e.g. obtaining a seven fold speedup on an eight core CPU.

### Acknowledgement

I would like to thank Thomas H. Westerdale, Ernesto Costa, Craig Shelden, David Oranchak, Kevin Sim, John Koza, Simon Waite and Gabriel Kronberger.