Fitness First

W. B. Langdon

Abstract With side effect free terminals and functions it is possible to evaluate the fitness of genetic programming trees from their parents without creating them. This allows selection before forming the next generation. Thus avoiding unfit runt Genetic Algorithm individuals, which will themselves have no children. In highly diverse GA populations with strong selection, more than 50% of children need not be created. Even with two parent crossover, in converged populations, e^{-2} = 13.5% can be saved. Eliminating bachelors and spinsters and extracting the smaller genetic material of each mating before crossover, reduces storage in an N multi-threaded implementation for a population M to $\leq$0.63M+N, compared to the usual M+2N. Memory efficient crossover achieves 692 billion GP operations per second, 692 giga GPops, at runtime on a 16 core 3.8GHz desktop.

Key words: Speedup technique, fast tree evaluation, memory efficient GA, generational EA, runt free broods, convergence, tournament selection, extended evolution, Long-Term Evolution Experiment, LTEE

1 Introduction

It is commonly held that genetic programming run time is dominated by the time to evaluate evolved individual program's fitness [7, 30]. However, in the last couple of years fitness evaluation for floating point problems has progressed enormously [10, 17, 15, 14, 11, 3], meaning in large programs of tens of millions of opcodes the primary cost can be in performing crossover rather than fitness evaluation, see Figures 1, 2 and 3. We show the cost of subtree crossover can be reduced by 1) doing

W. B. Langdon
Department of Computer Science, University College London, Gower Street, WC1E 6BT, UK e-mail: W.Langdon@cs.ucl.ac.uk
GPTP XVIII, W. Banzhaf, *et al.*, *Eds.*. 19-21 May 2021. Springer. *Preprint*

crossover after fitness and 2) separating the subtree donating parent (the dad). See Figures 4, 5 (page 6) 10 and 11 (page 11).

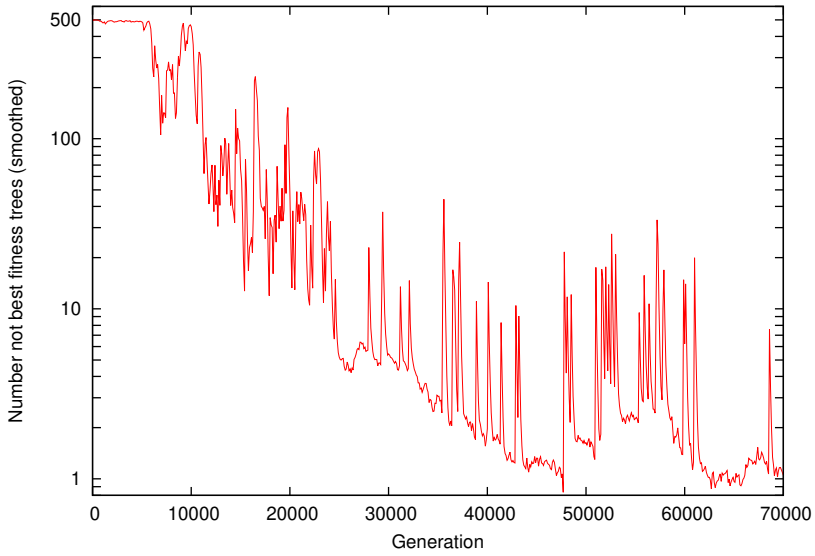

Fig. 1 Evolution of fitness convergence. Plot of number of individuals worse than the best smoothed by plotting running mean of 100 generations. Sudden upticks as new better individual is found and takes over the population. Pop=500.

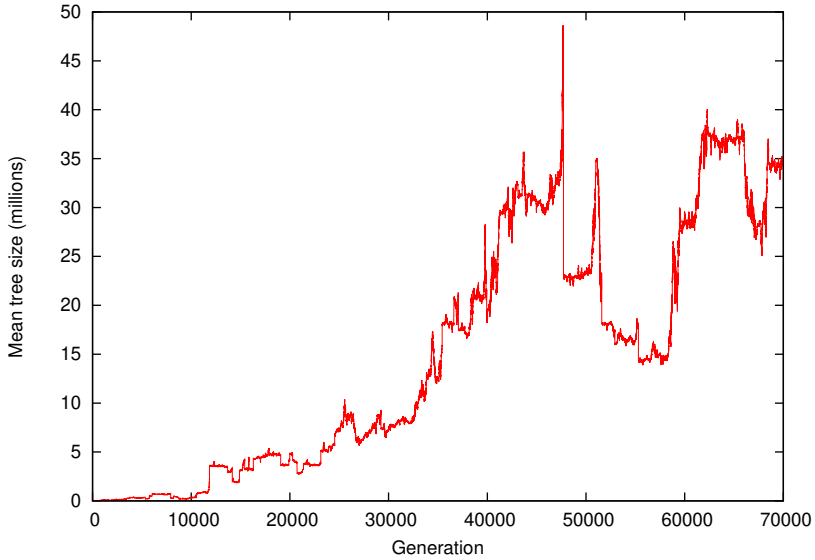

Fig. 2 Evolution of tree size.

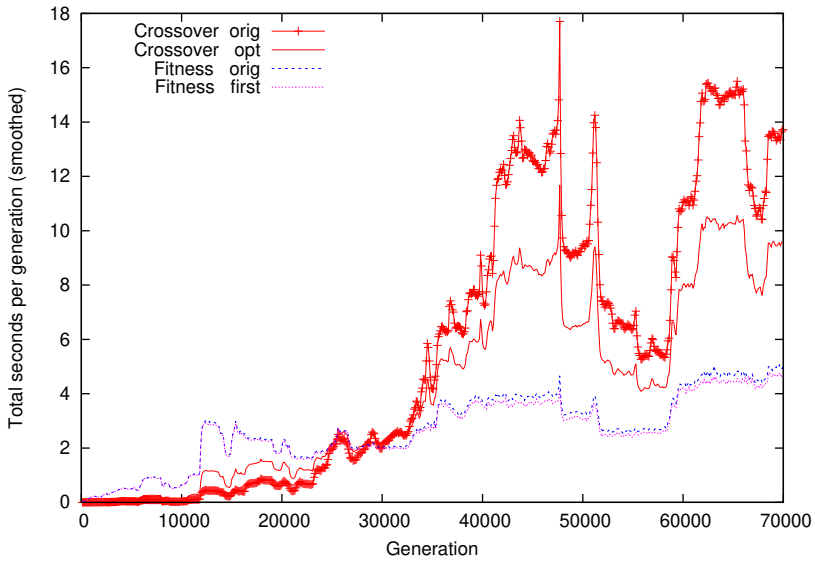

Fig. 3 Evolution of average time taken by incremental fitness first and crossover evaluation. Pop=500. 16 core Intel 3.80GHz i7-9800X. Running means of 100 generations.

The next section summarises recent use of high performance parallel computing for tree based genetic programming. This is followed by Section 3 which describes how it is possible to assign fitness values to the current generation before it is complete by incrementally evaluating [15] children using only the crossover points and their parents. Section 4 shows reversing the order of fitness and crossover allows us to avoid using crossover to create poor fitness individuals. Also separating subtrees from fathers eases other crossover optimisations, Section 4.1.

The final sections 5–8 deal with implementation issues and analysis. Section 5 says that, contrary to internet wisdom, current implementations of C++ memmove are not slow compared to memcpy and discusses its implications for our inplace crossover optimisation. Section 6 describes the GP's speed. Section 7 gives a brief model of the impact of tournament selection on diverse populations (such as those typically found near the start of GP runs). This complements the mathematical analysis in Section 3, which covers converged populations, when everyone has the same fitness. The two cases each have benefits which our crossover optimisations are able to exploit, leading to speedups both at the start and end of GP runs. Section 8 describes problems of load balancing to get peak performance from modern multi-core Intel CPUs before we conclude in Section 9. First we describe recent developments with speeding fitness evaluation and crossover using parallel hardware.

2 Faster Genetic Programming via Parallel Hardware

2.1 Multiple CPU Cores

Koza [7] described genetic programming as being embarrassingly parallel, in that by distributing the population, GP can easily be coded to get near 100% loading of parallel computers. Typically the population is spread across multiple computers which operate more or less independently. Similarly, our GP experiments are run on a parallel Intel multi-core desktop. There is a single administration thread, but with the creation of each individual in the population by crossover and also its incremental fitness evaluation being treated as separate tasks. These tasks are run in parallel by the hardware cores. The Linux posix pthreads environment is used with one thread per CPU core. Load balancing across the cores is achieved by each thread taking the next individual to be processed as it finishes the last, until the whole population has had its fitness calculated or the required members of the next population have been created using crossover.

This multi-threading strategy works well when the population size is much more than the number of CPU cores and the tasks are more or less the same size (but see Section 8.1) and means the population remains united. This approach also allows a light central core containing all the stochastic code with only resource intensive (deterministic) code running in parallel threads. Thus, with careful control of pseudo random number seeds, it makes it possible to replicate runs exactly in serial and different parallel environments. That is, a sequential run will produce the same sequence of populations as one using 8-cores, which in turn is the same as that produced on a 16-core machine. Indeed the system has been run on cluster nodes with 48 cores.

Note a single united panmictic population may converge more rapidly than in parallelisation schemes which require the population to be geographically divided between physically distinct processors. The next section considers a much finer grained parallelism in which fitness evaluation of a single individual is spread over up to 16 compute elements.

2.2 Multiple Fitness Cases Simultaneously

Our use of Intel's SIMD AVX-512 parallel vector instructions allows 16 test cases to be evaluated simultaneously [10, 16, 11]. This can be thought of as the floating point equivalent of Poli's sub-machine code GP [28]. With sub-machine code GP an opcode (e.g. AND) can be evaluated on 64 Boolean test cases at each clock tick [27, 9]. Indeed older AVX instructions have been used to evaluate 128 and 256 Boolean test cases simultaneously [6]. Also the newer AVX-512 instructions could be used to extend this to 512 test cases in parallel. Indeed genetic improvement ([32],[18, 24, 22, 25, 23]) has been applied to AVX code itself [11]. Our latest developments [15] mean in extended GP runs the primary cost is creating and storing the next generation, rather than calculating its fitness.

2.3 Fitness First

It is relatively straight forward to convert our bottom up incremental evaluation [15] from evaluating each child directly, to evaluating it indirectly via its parents, Figure 4. Thus we can find a child's fitness before creating the child. Figure 6 shows an example of incremental fitness evaluation using only the child's parents. Figure 7 shows an example from generation 1000 where incremental evaluation proceeds approximately half way from the crossover point to the root node. If it turns out the child is never used, e.g. because it is unfit or unlucky, it need not be created (Figure 5).

We assume the GP population is made of pure functions (i.e. there are no side effects) and the same test cases are used to assign fitness of the children as were used to find the fitness of their parents.

Fitness first starts by evaluating the subtree to be removed from the mum (white) and the subtree to be inserted (black), Figure 6. Apart from starting at the root of a subtree (i.e. within a parent) rather than at the root node, the evaluation is the same as usual. I.e., the normal depth first recursive evaluation is used for all subtrees that have to be evaluated. (Albeit if AVX-512 is supported in hardware, we use parallel AVX instructions.)

If, for all test cases, the values produced by the new code to be inserted are identical to those produced by the code to be removed, the inserted code has no effect and the child's fitness must be the same as the mum's. If any are different, we proceed up the mum tree towards its root.

The example in Figure 6 shows the next node up is a plus. We find the other subtree in the mum that is the plus' other argument (shown in red) and recursively evaluate it for all the test cases. Again this GP code (which must be identical to that in the as yet unborn child) is run in the mum *in situ*. The evaluation again gives a vector of floats (one element per test case). Next the function (plus) is applied to each value in the vector (red arrow) and the corresponding value from the mum subtree (light blue arrow) and similarly to the values from child's subtree (black arrow). This gives us two float vectors (one for mum and one for the child). Again if they are equal we can stop, since, if they are equal, they would remain equal all the way to the root node. And therefore the child's fitness must be equal to that of its mum. Note we still have not gone near the child and indeed we have finished with the dad.

If the two vectors are not identical, we proceed up the mum tree evaluating side subtrees and nodes on the path to the root until either we reach a point where the values in the mum and the values the child would have been identical or we reach the root. If we reach the root, the child's fitness is calculated from the values in its vector of evaluations for each test case. Again we do not need to create the child to do this.

In very big trees, populations are often highly converged and children often inherit the same fitness values as their parents. In which case, fitness first evaluation can give orders of magnitude savings in evaluation time.

Table 1 gives details of our GP.

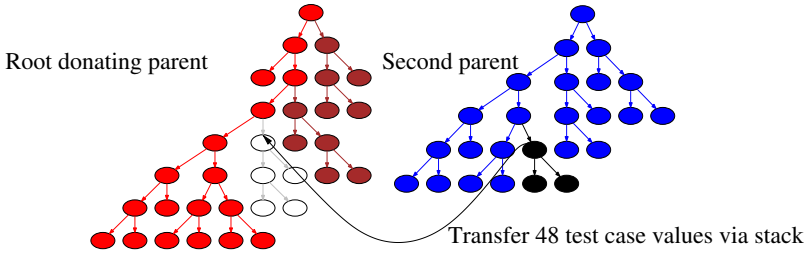

Fig. 4 Fitness is evaluated using only parents, i.e., before the child is created by crossover. Assuming no side effects, the subtree to be inserted (black) is evaluated on all test cases and values are transferred to evaluation of mum (left) at the location of the subtree to be removed (white). We use our incremental evaluation [15], so differences between original code (white subtree) and new are propagated up 1st parent (mum) until either all differences are zero or we reach the root node.

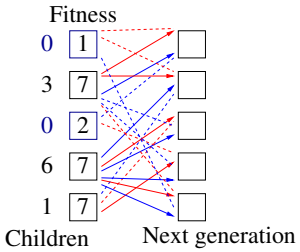

Fig. 5 As fitness can be calculated before crossover (Figure 4), the parents can be chosen before crossover too. Here two low fitness individuals (fitness 1 and 2) have no children and hence their creation need not be completed. Lines indicate the two members of each tournament used to select the first (red) and second (blue) parent. Solid lines with arrows are the winners of each tournament [29]. (Binary tournament only for illustration, we actually use tournaments with 7 members.) All common EA selection schemes (with either mutation or crossover) are guaranteed to have members of the current population who will not have children in the next generation.

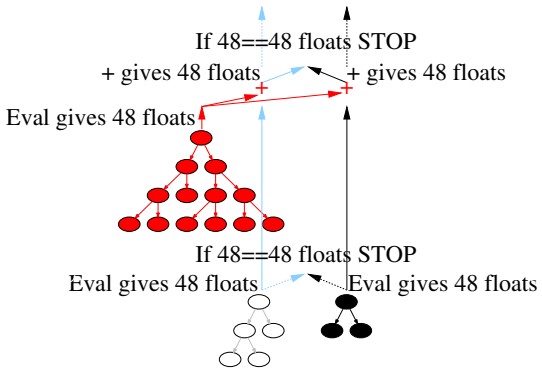

Fig. 6 "Fitness first" begins by evaluating the subtree to be removed from the mum (white) and the subtree to be inserted (black). It proceeds up the mum's tree until either the evaluation in the mum and unborn child are the same or it reaches the root node. The red subtree is in the mum but it is identical to the code in its child and so need be evaluated only once per test case. Note the code from the parents is evaluated without creating the child. Example from Figure 4. See also Figure 7.

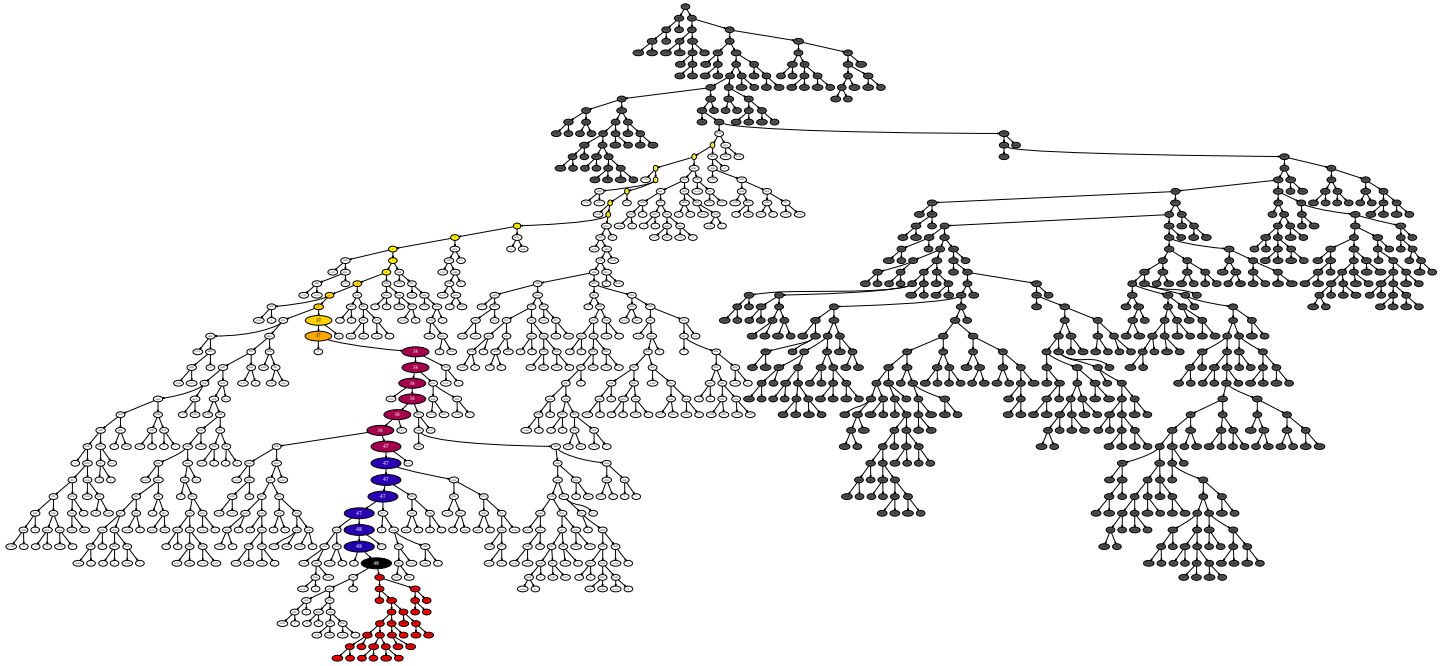

Fig. 7 Example of incremental evaluation [15]. Parent tree is modified by crossover replacing code with inserted subtree (red). Replaced and new code are both evaluated on the test set (48 tests). As they are different, the next node above the crossover point is evaluated, taking the 48 values returned by the original and new code (together with its other argument from the unchanged code). Here too evaluation in the parent and (putative) child are different, so evaluation proceeds up the tree towards its root node (see also Figure 6). The chain of evaluated nodes is in colour [19]. The size and numbers in each node gives the number of test cases where the evaluation of the parent and (putative) child are not identical. Their average evaluation difference is indicated on a log scale by the node's colour. Average differences greater than 0.01 are shown with dark colours, less than 0.01 by brighter colours. Brightest yellow shows smallest non-zero difference (RMS 3.1 10^{-10}). If, as here, parent and child evaluations are identical before reaching the root node, the remainder of the evaluation is not needed (gray nodes) and is skipped and instead fitness is copied from the parent.

Table 1 Evolution of Sextic polynomial [7] symbolic regression binary trees using GPquick's one byte per opcode.

Terminal set:	X, 250 constants between -0.995 and 0.997
Function set:	MUL ADD DIV SUB
Fitness cases:	48 fixed input -0.97789 to 0.979541 (randomly selected from -1.0 to +1.0). For simplicity, we use all the same test cases in each generation, although of course, testing can be reduced [21, 5] or made dynamic [18] Target $y = xx(x-1)(x-1)(x+1)(x+1)$
Selection:	Tournament size 7 with fitness = $\frac{1}{48}\sum_{i=1}^{48} \lvert GP(x_i) - y_i \rvert$
Population:	500 binary trees. Panmictic (fully mixed), non-elitist, distinct (non-overlapping) generations.
Parameters:	Initial population ramped half and half [7], depth between 2 and 6. 100% unbiased subtree crossover. 70 000 generations

3 Avoiding Effort Wasted on Poor Fitness Individuals

Whereas the previous approaches, described in Sections 2.1 and 2.2, speed up genetic programming by use of more powerful hardware, we have implemented a fitness first scheme which speeds up GP by 14% by doing less work. (Fitness first could be widely applicable in evolutionary computing, however only when constructing members of the population is expensive compared to fitness evaluation is it likely to be useful.) For simplicity our implementation ensures that it produces identical results. That is, given the same pseudo random number seed, the population at each generation in the new implementation is identical to that given before.

Early in GP runs at each generation many poor individuals are created (see Figure 8). All Evolutionary Algorithm (EA) selection schemes aim to ensure poor individuals are less likely to be selected to have children themselves. (See example with a population of five in Figure 5.) Since childless individuals have no impact on the future course of the run, it is wasteful to create such individuals.

Apart from Baker's Stochastic Uniform Selection (SUS) [1], commonly used selection schemes, such as tournament selection, allocate children independently. Thus, even later in the run, when many programs have the same fitness, there will be some parents who by chance get more than the average number of children and some who get less. With two parent crossover, on average each member of the current population gets two children. In the limit of large converged populations (containing M individuals) on average there will be $e^{-2}M$ individuals which are never selected to have children (see right hand side of Figure 8). If we consider just the first parent in crossover, or 100% one parent mutation, then this rises to $e^{-1}M$.

As Figure 8 shows, delaying crossover until after fitness selection can save creating more than half the population during the early part of a run. Even later, when convergence ensures almost the whole population has the same fitness, 14% (e^{-2}) of the population need not be created. With very large trees, run time can be dominated by crossover (see Figure 3), thus run time savings are possible by avoiding complete generation of poor fitness individuals.

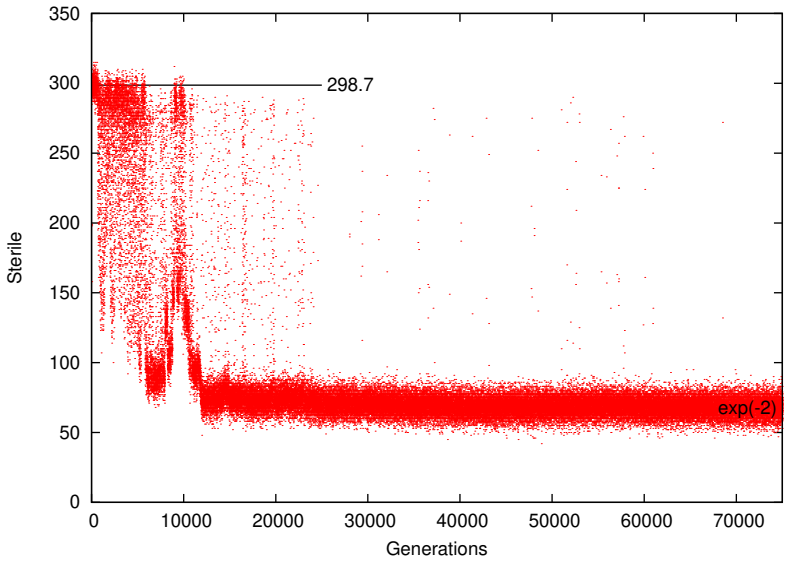

Fig. 8 Evolution of number in population without children in next generation. 100% two parent crossover, 7-tournament, pop=500.

4 Asymmetry of GP subtree crossover

We use Koza's two point subtree crossover [7] but for simplicity with both crossover points chosen uniformly at random. That is, we do not include a bias in favour of internal nodes.

Figure 9 shows the dramatic imbalance in the contributions of the two programs chosen to be parents for the new individuals (note log scale). For example, in generation 15 000 the root donating trees (mums) supply more than a thousand times as many opcodes as the dads.

The lower (red) solid line in Figure 9 plots the running mean smoothed over 100 generations of the number of inserted opcodes from each dad program. After generation 15 000 it changes little, and averages 275.4 opcodes. However the distribution of inserted subtree sizes varies widely in each generation and between generations (blue dots). It has a long tail with the mean being typically more than three times the median. The dad long tailed distribution has some impact on run time, with some trees taking far longer to evaluate for fitness than others, making it harder to distribute work evenly between threads on multi-core CPUs. (Section 8.1 considered how often cores are not being used.) In contrast the number of opcodes inherited from mum (top line in Figure 9) closely follows the total tree size and even after generation 15 000 continues to bloat.

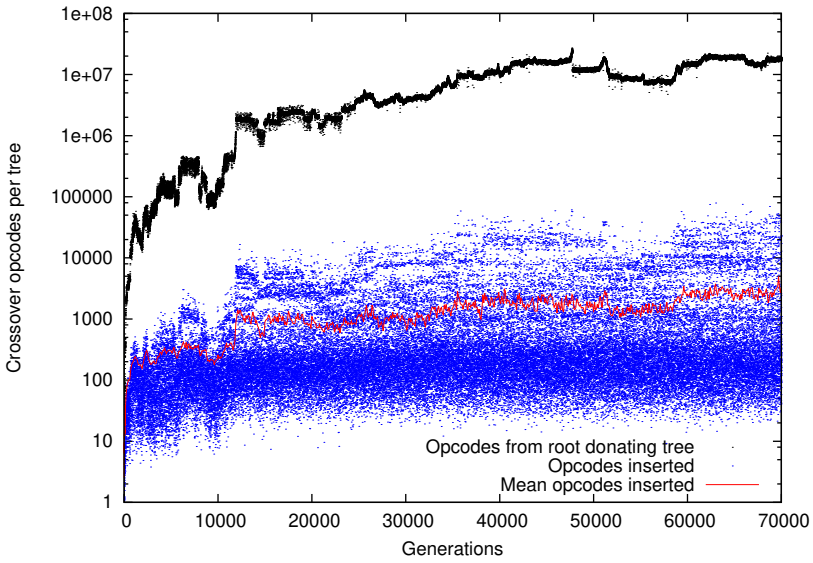

Fig. 9 Evolution of number opcodes from each parent. Mums top line. Dads blue lower cloud. Note log vertical scale.

4.1 Last Child Inplace Dad-Less Crossover

Initially the populations are very variable and, with strong selection, breeding is concentrated in a few fit parents. As the populations starts to converge, there are more parents (with fewer children each). In each generation, as each child is created, eventually for each parent, there is only one child left to be created. (Locks are used to ensure multi-threaded code neither skips anyone nor creates any child twice). On reaching the last child for a root donating parent, instead of copying the code into the child (see Figure 10), the buffer holding the parent's genome is unhooked from the parent and passed to the child. This saves copying the first part of the child (see Figure 11).

As we saw in Figure 9, the second parent (dad) donates only a tiny fraction of the opcodes in the child. Therefore we extract and save all the subtrees which will be inserted later. This is relatively cheap and is done (in the sequential code) before the bulk of the crossover operations are done using the root donating parents (mums) in multi-threaded code. This simple step allows the mum's last child crossover short cut (Figure 11) to be used about twice as often.

Notice whilst fitness convergence reduces the number of childless members of the population (Section 3), here it helps: as spreading the breeding effort, means there are more parents in general, and thus more cases where a mum has only one child left to be created. That is, convergence increases the number of times inplace crossover optimisation can be applied. Figure 12 shows later in the run as the population converges and there are more parents with children, the number of inplace crossovers rises, so that on average 268.1 ($\lesssim M(1-e^{-2})(1-e^{-1})$) crossovers are done inplace per generation.

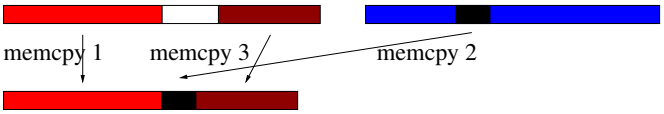

Fig. 10 Andy Singleton's GPquick [31] subtree crossover requires three memcpy buffer copies: 1) root segment of donating parent (mum, red/brown) is copied to offspring buffer. 2) subtree from second parent (dad, blue/black) is copied to offspring. 3) tail (brown) of 1st parent copied to child.

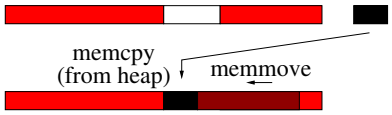

Fig. 11 Inplace subtree crossover. Offspring is last child of 1st parent and reuses its buffer. Only subtree to be inserted (black) of 2nd parent (dad) is kept. 1) Dad subtree overwrites mum's buffer. 2) In 71% of children the subtree to be remove (white) and to be inserted (black) are different sizes, and so memmove is used to shuffle the second part of mum's buffer (brown) up or down.

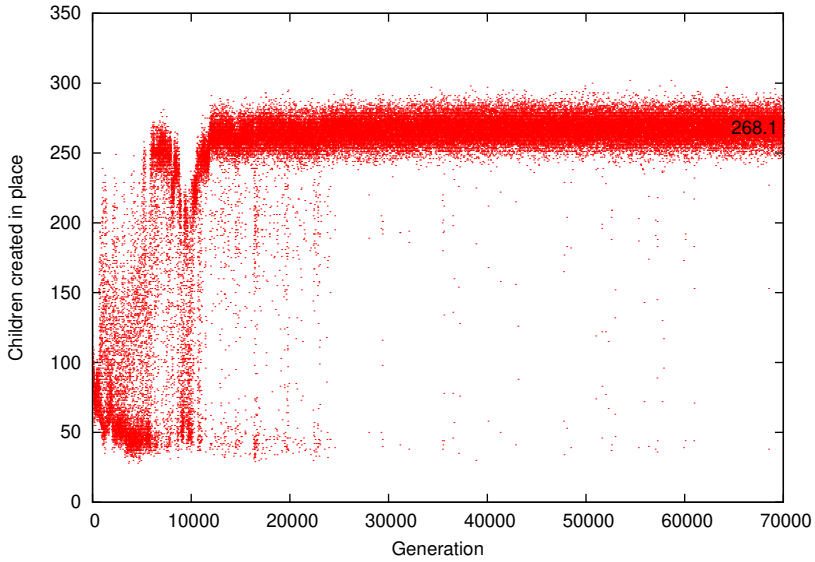

Fig. 12 Number of times per generation when creating non-sterile children in the next, the root donating parent (mum) has only one more child to create and so crossover can reuse part of its genome. Pop=500. See Section 4.1 and Figure 11.

In about one third (28.9%) of cases, the removed subtree and inserted subtree are the same size. If so, the mum's buffer can be simply over written with the inserted code (from the dad). However most (71.1%) of the time they are not the same size and the buffer must be shuffled either up or down to take account of the difference in the subtree sizes (see Figure 11). This shuffling is done using memmove, rather than memcpy. (See also Section 5). Figure 12 confirms, by excluding the dads from crossover, we can use the inplace short cut more than half the time.

The large blue cloud in Figure 13 shows the time originally taken by each of 16 threads to perform crossover of the whole of the current generation late in the run. The tight red cluster of dots show the same populations after crossover has been optimised to: 1) ignore individuals which will not have children (saving about 13.5%) and 2) where possible, modifying chromosomes inplace. Figure 13 confirms we are reducing the volume of opcodes copied by crossover by almost a half (48.1%). This leads to a reduction in the total time taken by the crossover threads by about a quarter (24.4%).

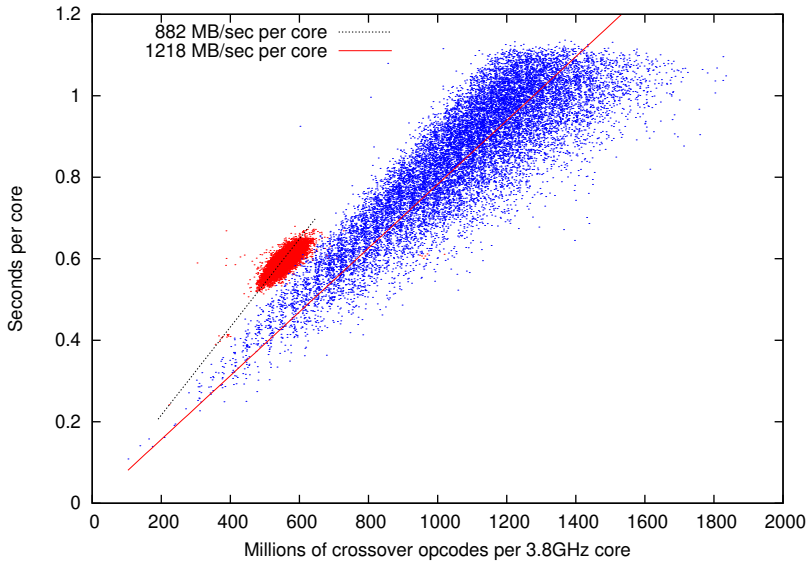

Fig. 13 Time per thread to create children using fatherless (left, red) and traditional (right, blue) crossovers v. the number of opcodes the thread processes (see Section 4.1). To reduce clutter just generations 69 000–70 000 are plotted. 16 core 3.8GHz desktop.

5 Efficiency of memmove v. memcpy

Although much has been made of the efficiency of memcpy compared to that of memmove, with the GCC 9.3.1 g++ compiler and version 2.17 of the GNU C run time library, for our new crossover implementation we found little difference (see Figure 14). Indeed instrumenting the memmove operation and the corresponding memcpy, shows memmove to be 14% faster. On average at the end of the run memmove moves 970MB/second per core while memcpy copies 851MB/sec per core (on a 3.80 GHz Intel i7-9800X desktop). Note that these are in place measurements, rather than standalone benchmarks and so memmove has on average slightly more work.

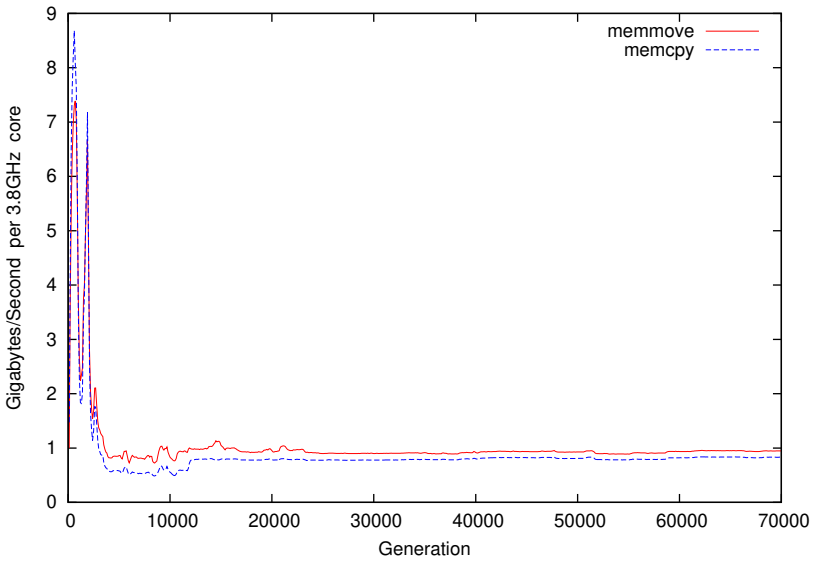

Fig. 14 Evolution of speed of memmove and memcpy as used in GPquick crossover. It appears the initial high speed of both is due to GP trees not exceeding the cache size, 16.5MB. Plots are smoothed running means of 100 generations. Note traditionally bandwidth counts each byte moved or copied twice, i.e. a byte into the CPU and a byte out to memory.

6 Speed of Fitness First and Incremental Fitness

As described in Section 2.3 (page 5), our incremental fitness evaluation [15], which evaluates side-effect free trees from the crossover point towards the root, can be readily adapted to evaluate the child via its parents. Apart from adapting pointers to the crossover points in the parents, rather than in their child, little is changed. As expected, Figure 15 shows the time taken to find the fitness of the whole of the current generation depends linearly on the number of opcodes that have to be evaluated. Note inparticular moving from incremental evaluation of the children to evaluating them by using only their parents has made little difference, see lower dash and dotted traces in Figure 3 on page 3. (The fitness results are of course identical.)

7 Mathematical Model of Number of Parents

Section 3 (page 8) has already shown a model of crossover which predicts the number of members a population with near uniform fitness which do not have children in the next generation will be $e^{-2}M$. Figure 8, page 9, confirms the model essentially holds after generation 15 000 even though there remain a few members of the population with an atypical fitness value. (See also Figure 1 on page 2.)

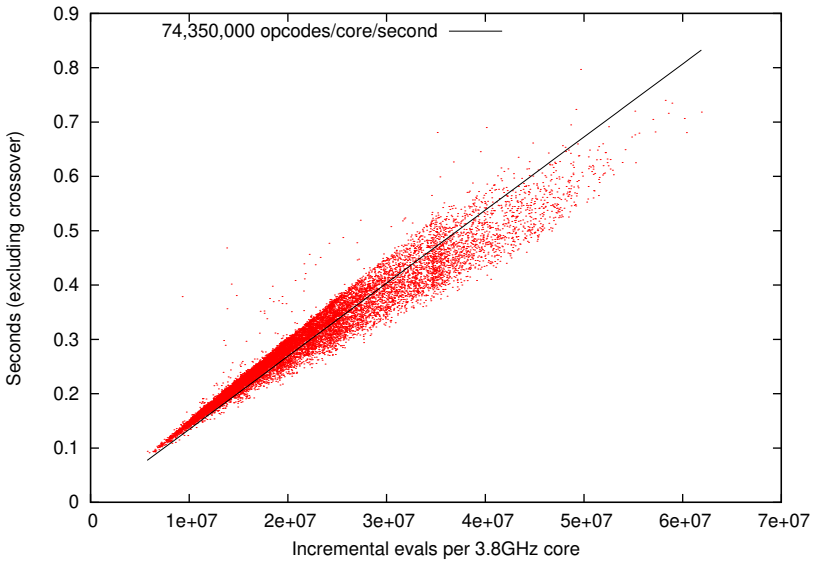

Fig. 15 Time taken each generation by each thread to calculate fitness against the number of opcodes the thread processes. Note incremental fitness evaluation using the child's two parents before the child is created. Scatter plot, 16 threads, generations 69 000 to 70 000.

7.1 Number of Parents Initially and in Diverse Populations

Where there is a fitness gradient across the population, a wide variety of selection schemes will allocate children to the best members of the population. This means even with two parent operations, like crossover, there will be many low fitness or just unlucky members of the population, whose genetic material will be lost.

Goldberg's selection pressure [4] of commonly used fitness selection schemes has been mathematically analysed by Blickle [2], and ourselves [20, page 185] giving, in a diverse population, the chance of the r^{th} best individual in the population winning the next tournament as $(r/M)^T - ((r-1)/M)^T$ (see Figure 16). Assuming distinct non-elitist generations and T-tournament selection, on average the best member of the population will be selected to be a first parent T times. Using crossover there are two parents, so parents have twice as many children. Thus, the best in the population has on average $2T = 14$ children (see left of Figure 16). Even in modest population sizes, the worst member of the population is unlikely to have children.

A Monte Carlo simulation predicts almost 60% of random populations with a tournament size of seven will not have children, see Figure 16. This is is good agreement with many populations up to about generation 5000, i.e. before they near fitness convergence (see 298.7 of 500, in Figures 8 and 16).

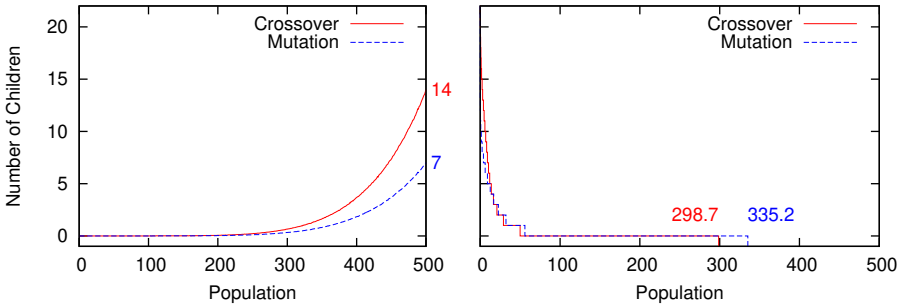

Fig. 16 Left: Offspring v. rank. Expected number of children with tournament size T=7 in initial diverse populations [2, 20]. Pop=500. Single parent mutation (dashed line) not used but shown for comparison. Right: same data as histograms. E.g. on average 298.7 members of the population (with crossover) have no children, 49.9 have one child, 29.1 two and so on.

8 Multi-threading Implementation Issues

To minimise memory consumption, we process children whose parents have only one child left be delt with before the others [12]. This avoids having to store both the current and the next population at the same time. As children are created, their parents are moved between two queues. One queue is for parents with one child left to process and another queue is for parents with two or more children yet to be created. When a parent's last child has been created, the parent can be deleted and the memory it occupied can be freed and thus be used by new children in the next generation. As we reported earlier [12], with the usual crossover and fitness evaluation order, M+2N memory buffers are needed. Where M is the population size and there are N threads. The factor of two comes from using two parent crossover. (If using only single parent mutation, M+N memory buffers would be needed.)

By using fatherless crossover, M+2N, can be reduced to M+1N. Although fatherless crossover, Sections 4 and 4.1, does require storing the subtrees to be inserted on the heap. However typically the opcodes inherited from the dads occupy less than a megabyte (see Figure 9 on page 10).

The two multi-threaded queues [12] give an easy way of recognising mums with only one child left to create and so help implementing inplace crossover, see Section 4.1 (page 10) and Figures 10 11 (page 11). Also, as inplace crossover automatically shares the memory used by the parent and the offspring, in practice memory consumption is reduced to approximately $(1-e^{-1})$M+N = 0.63M+N. That is, although we still have to allow for N threads operating simultaneously: population fitness convergence, not creating low fitness individuals who will not have children, fatherless crossover and inplace crossover, together (as well as speeding up GP) reduce memory consumption by about a third.

Although we know on which of the two queues parents must be placed [12], we are still free to decide where in the given queue they are to be. As yet we have not exploited this ordering freedom. In future there may be modest saving to be made by better scheduling work between the available threads. (We return to this in Section 8.2.)

8.1 Idle Threads

Figures 17 and 18 show the total thread idle time on a 16 core desktop. Figure 18 shows the average waiting time as a fraction of the elapse time for each set of 16 threads in that generation. To improve visibility, the plots have been smoothed by taking running averages over 100 generations.

In the original scheme (blue dashed lines) multiple threads performed crossover and evaluated fitness [15]. I.e. children were created and their fitness was immediately calculated, as an indivisible unit, by the same thread. (Note crossover was performed to create 100% of each population.) In the new scheme, crossover of only the part of the *next* generation which has children is done (red lines with crosses). Fitness evaluation is unchanged. Since crossover and fitness now operate on different individuals, they are separated, and each is done by their own set of threads. For simplicity the two sets do not overlap. I.e. the fitness threads synchronise together and then the crossover threads synchronise together. In principle the two types of threads could be intermingled, but this would complicate the implementation.

Thus, in the original scheme, there is only one synchronisation point at the end of each generation, where idle threads are forced to wait. Whereas there are two synchronisation points in the new scheme. (Hence the three sets of lines in Figures 17 and 18.)

In both schemes, the later stages of the run are dominated by the crossover time (see top two lines in Figure 3 page 3). However crossover time is much more predictable and uniform than the time to do fitness evaluation (where the longest fitness evaluation can exceed the average by a factor of 100 or more). Fitness evaluation is simply scheduled by the next free thread taking the next individual. Whereas the order of the crossover threads is dictated by Koza's algorithm to minimise buffer usage [8, pages 1044-1045], [12], [13] (see previous section).

The more uniform duration of the crossover tasks means thread idle time, as a fraction of total time (Figure 18), is low. The wide variation in fitness evaluation time leads to proportionately more wasted thread idle time. However this is mitigated in bloated runs by the great speed of incremental fitness evaluation compared to the time taken to create enormous trees. For example, on average over the last 100 generations, GP was unable to use 39%, of the 16 core computer during fitness evaluation (top trace in Figure 18), whilst for the new crossover it was 1% unused.

8.2 Future Work: Predicting Thread Execution Time

As mentioned in the previous section, when a thread finishes a task it takes the next free task and begins processing it. Idle time comes from threads running out of tasks at different times. When tasks take different lengths of time, there may be practical savings from more proactive scheduling. Since the threads are (assumed to be) homogeneous, a simple heuristic of starting with the longest tasks (spread across

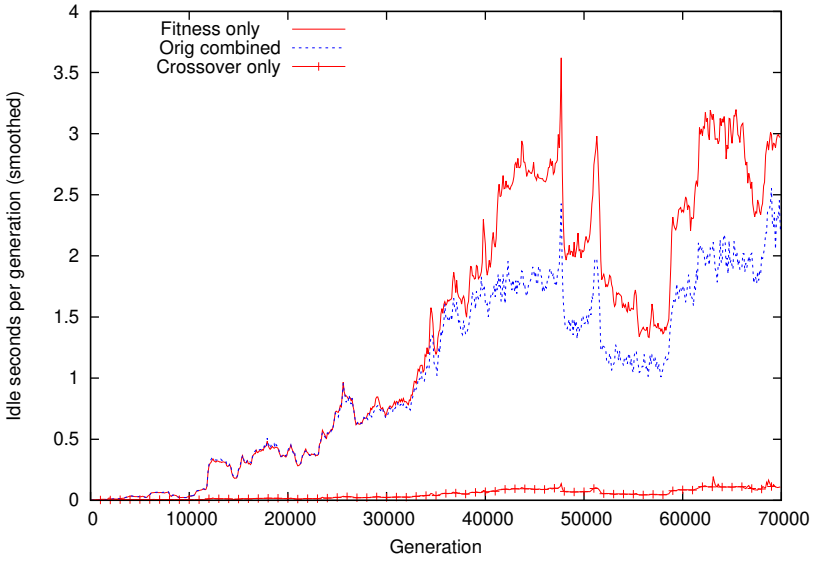

Fig. 17 Total time spent by 15 threads waiting for the slowest to synchronise per generation (on 16 core 3.8GHz desktop). In the original implementation (dashed blue line) the original crossover and our incremental [15] fitness evaluation were performed together. In the new crossover and fitness are separated, leading to two synchronisation steps per generation and two sets of idle threads (solid red lines). See also Figure 18.

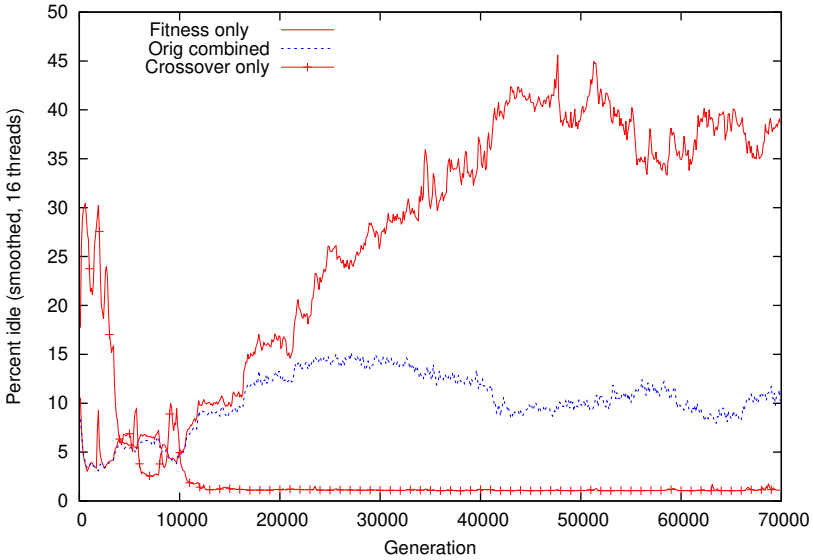

Fig. 18 Time spent by 15 threads waiting for the slowest as a fraction of time taken by all 16. Data as Figure 17 but expressed as percentages.

all the threads) and then moving to progressively shorter tasks, may be sufficient. E.g. sort the tasks into execution time order and then run as now.

Crossover time can be readily predicted from the amount of memory to be moved (memmove) or copied (memcpy). Given the size of individuals and the location of crossover points, both can be calculated in advance. So, for simplicity treating memmove and memcpy as the same, to minimise idle time, we might want to order the crossover queues to put the largest children first. However, to maximise runtime savings from inplace crossover, we might want to try to schedule crossovers so that children with the largest root segments are the last to be done for their mums. Alternatively to save memory, we might want to do them as soon as possible. (In [12] we treated all trees as being the same size.)

Fitness evaluation time is very variable and hard to predict, as, even though it is proportional to the number of opcodes to be evaluated (Figure 15), the number of evals is only known after the evaluation. It may be possible even a crude model might help. E.g. guess that a large (or very different) subtree to be inserted, will cause more disruption and hence require more evals, than a smaller or more similar one. Fitness first execution times can be very variable and, with 16 threads, a single evaluation can take as long as the rest of the population (spread over 15 threads). Given this, there may be only marginal gain from clever scheduling. As the variation gets still bigger it might be, for very time consuming individuals, worthwhile to spread their fitness evaluation across multiple threads.

9 Conclusions

Although we have couched our work in GP terms, the memory savings hold for evolutionary algorithms with crossover or with mutation alone. Where EA chromosomes are enormous and (changes in) fitness can be quickly calculated, these ideas of reversing the order of fitness calculation and offspring creation, might also be beneficial.

For a typical small GP population (500 trees) on a 16 node desktop, memory use can be reduced by about a third. On that desktop we have performance *equivalent* to 692 Giga GPop/s ($6.92\ 10^{11}$ GP operations per second) which is more than four times the performance that we claimed as a record [16] for a single computer GP system and that was a 48 core cluster server.

We have shown it is practical to delay subtree crossover until after fitness evaluation and so only create GP trees which themselves will carry genetic material into subsequent generations. Typically early in GP runs, tournament selection gives a very high selection pressure, meaning there are many trees of low fitness which do not have children. In any evolutionary algorithm, by reversing the usual order of program evaluation and creation, it is no longer necessary to create low fitness individuals. This can save a large fraction of the memory to store them. Even later in GP runs, when fitness convergence may spread children more evenly, and the cost

of creating new GP trees may exceed the cost of fitness evaluation, the saving can be worthwhile.

Even when trees are large, the asymmetry of GP subtree crossover means, the code to be inserted into the next generation, (i.e. all the subtrees from each father) is small. Indeed it may fit into fast cache memory. These subtrees can be extracted from the population before the bulk crossover operations. This simplifies the rest of the crossover operations, as they now only use one parent from the population (i.e. they are fatherless). This can be beneficial in terms of freeing memory early and reducing crossover effort.

The new GPQuick code is available in http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPinc.tar.gz

Acknowledgements I would like to thank Stephan Winkler, Sara Silva, people at GPTP and anonymous reviewers. This work was inspired by conversations at Dagstuhl Seminar 18052 on Genetic Improvement of Software [26].

Funded by EPSRC grant EP/P005888/1.

References

1. Baker, J.E.: Reducing bias and inefficiency in the selection algorithm. In: J.J. Grefenstette (ed.) Proceedings of the Second International Conference on Genetic Algorithms and their Application, pp. 14–21. Lawrence Erlbaum Associates, Cambridge, MA, USA (1987)
2. Blickle, T.: Theory of evolutionary algorithms and application to system synthesis. Ph.D. thesis, Swiss Federal Institute of Technology, Zurich, Switzerland (1996). URL http://dx.doi.org/10.3929/ethz-a-001710359
3. de Melo, V.V., Fazenda, A.L., Sotto, L.F.D.P., Iacca, G.: A MIMD interpreter for genetic programming. In: P.A. Castillo, J.L. Jimenez Laredo, F. Fernandez de Vega (eds.) 23rd International Conference, EvoApplications 2020, *LNCS*, vol. 12104, pp. 645–658. Springer Verlag, Seville, Spain (2020). URL http://dx.doi.org/10.1007/978-3-030-43722-0_41
4. Goldberg, D.E.: Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley (1989)
5. Guizzo, G., Petke, J., Sarro, F., Harman, M.: Enhancing genetic improvement of software with regression test selection. In: A. van Deursen, T. Xie, N.J.O. Dieste (eds.) Proceedings of the International Conference on Software Engineering, ICSE 2021. IEEE (2021). URL http://dx.doi.org/10.1109/ICSE43902.2021.00120. Winner ACM SIGSOFT Distinguished Artifact Award
6. Hrbacek, R., Sekanina, L.: Towards highly optimized cartesian genetic programming: from sequential via SIMD and thread to massive parallel implementation. In: C. Igel, D.V. Arnold, C. Gagne, E. Popovici, A. Auger, J. Bacardit, D. Brockhoff, S. Cagnoni, K. Deb, B. Doerr, J. Foster, T. Glasmachers, E. Hart, M.I. Heywood, H. Iba, C. Jacob, T. Jansen, Y. Jin, M. Kessentini, J.D. Knowles, W.B. Langdon, P. Larranaga, S. Luke, G. Luque, J.A.W. McCall, M.A. Montes de Oca, A. Motsinger-Reif, Y.S. Ong, M. Palmer, K.E. Parsopoulos, G. Raidl, S. Risi, G. Ruhe, T. Schaul, T. Schmickl, B. Sendhoff, K.O. Stanley, T. Stuetzle, D. Thierens, J. Togelius, C. Witt, C. Zarges (eds.) GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation, pp. 1015–1022. ACM, Vancouver, BC, Canada (2014). URL http://dx.doi.org/10.1145/2576768.2598343

7. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992). URL http://mitpress.mit.edu/books/genetic-programming
8. Koza, J.R., Andre, D., Bennett III, F.H., Keane, M.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann (1999). URL http://www.genetic-programming.org/gpbook3toc.html
9. Langdon, W.B.: Long-term evolution of genetic programming populations. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '17, pp. 235–236. ACM, Berlin (2017). URL http://dx.doi.org/10.1145/3067695.3075965
10. Langdon, W.B.: Parallel GPQUICK. In: C. Doerr (ed.) GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion, pp. 63–64. ACM, Prague, Czech Republic (2019). URL http://dx.doi.org/10.1145/3319619.3326770
11. Langdon, W.B.: Genetic improvement of genetic programming. In: A.S. Brownlee, S.O. Haraldsson, J. Petke, J.R. Woodward (eds.) GI @ CEC 2020 Special Session, p. paper id24061. IEEE Computational Intelligence Society, IEEE Press, internet (2020). URL http://dx.doi.org/10.1109/CEC48606.2020.9185771
12. Langdon, W.B.: Multi-threaded memory efficient crossover in C++ for generational genetic programming. SIGEVOLution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation **13**(3), 2–4 (2020). URL http://dx.doi.org/10.1145/3430913.3430914
13. Langdon, W.B.: Multi-threaded memory efficient crossover in C++ for generational genetic programming. ArXiv (2020). URL http://arxiv.org/abs/2009.10460
14. Langdon, W.B.: Fitness first and fatherless crossover. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '21. ACM, Internet (2021). URL http://dx.doi.org/10.1145/3449726.3459437. Forthcoming
15. Langdon, W.B.: Incremental evaluation in genetic programming. In: T. Hu, N. Lourenco, E. Medvet (eds.) EuroGP 2021: Proceedings of the 24th European Conference on Genetic Programming, *LNCS*, vol. 12691, pp. 229–246. Springer Verlag, Virtual Event (2021). URL http://dx.doi.org/10.1007/978-3-030-72812-0_15
16. Langdon, W.B., Banzhaf, W.: Continuous long-term evolution of genetic programming. In: R. Fuechslin (ed.) Conference on Artificial Life (ALIFE 2019), pp. 388–395. MIT Press, Newcastle (2019). URL http://dx.doi.org/10.1162/isal_a_00191
17. Langdon, W.B., Banzhaf, W.: Faster genetic programming GPquick via multicore and advanced vector extensions. Tech. Rep. RN/19/01, University College, London, London, UK (2019). URL http://www.cs.ucl.ac.uk/fileadmin/user_upload/avx_rn1901.pdf
18. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation **19**(1), 118–135 (2015). URL http://dx.doi.org/10.1109/TEVC.2013.2281544
19. Langdon, W.B., Petke, J., Clark, D.: Dissipative polynomials. In: N. Veerapen, K. Malan, A. Liefooghe, S. Verel, G. Ochoa (eds.) 5th Workshop on Landscape-Aware Heuristic Search, GECCO 2021 Companion. ACM, Internet (2021). URL http://dx.doi.org/10.1145/3449726.3463147
20. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer-Verlag (2002). URL http://dx.doi.org/10.1007/978-3-662-04726-2
21. Lim, M., Guizzo, G., Petke, J.: Impact of test suite coverage on overfitting in genetic improvement of software. In: J.P. Galeotti, B. Sharif (eds.) 12th International Symposium on Search Based Software Engineering SSBSE 2020, *LNCS*, vol. 12420, pp. 188–203. Springer, Bari, Italy (2020). URL http://dx.doi.org/10.1007/978-3-030-59762-7_14
22. Petke, J.: Constraints: The future of combinatorial interaction testing. In: 2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing, pp. 17–18. Florence (2015). URL http://dx.doi.org/doi:10.1109/SBST.2015.11
23. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Transactions on Evolution-

ary Computation **22**(3), 415–432 (2018). URL http://dx.doi.org/doi:10.1109/TEVC.2017.2693219

24. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: M. Nicolau, K. Krawiec, M.I. Heywood, M. Castelli, P. Garcia-Sanchez, J.J. Merelo, V.M. Rivas Santos, K. Sim (eds.) 17th European Conference on Genetic Programming, *LNCS*, vol. 8599, pp. 137–149. Springer, Granada, Spain (2014). URL http://dx.doi.org/10.1007/978-3-662-44303-3_12
25. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Specialising software for different downstream applications using genetic improvement and code transplantation. IEEE Transactions on Software Engineering **44**(6), 574–594 (2018). URL http://dx.doi.org/10.1109/TSE.2017.2702606
26. Petke, J., Le Goues, C., Forrest, S., Langdon, W.B.: Genetic improvement of software: Report from dagstuhl seminar 18052. Dagstuhl Reports **8**(1), 158–182 (2018). URL http://dx.doi.org/10.4230/DagRep.8.1.158
27. Poli, R.: TinyGP. TinyGP GECCO 2004 competition (2004). URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/poli04__tinyg.pdf
28. Poli, R., Langdon, W.B.: Sub-machine-code genetic programming. In: L. Spector, W.B. Langdon, U.M. O'Reilly, P.J. Angeline (eds.) Advances in Genetic Programming 3, chap. 13, pp. 301–323. MIT Press, Cambridge, MA, USA (1999). URL http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch13.pdf
29. Poli, R., Langdon, W.B.: Running genetic programming backward. In: T. Yu, R.L. Riolo, B. Worzel (eds.) Genetic Programming Theory and Practice III, *Genetic Programming*, vol. 9, chap. 9, pp. 125–140. Springer, Ann Arbor (2005). URL http://dx.doi.org/10.1007/0-387-28111-8_9
30. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008). URL http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza)
31. Singleton, A.: Genetic programming with C++. BYTE pp. 171–176 (1994). URL http://www.assembla.com/wiki/show/andysgp/GPQuick_Article
32. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation **15**(4), 515–538 (2011). URL http://dx.doi.org/10.1109/TEVC.2010.2083669