THE PROBLEM-DEPENDENT NATURE OF PARALLEL PROCESSING IN GENETIC PROGRAMMING

William F. Punch
Michigan State University GARAGe
A714 Wells Hall, East Lansing MI, 48824
punch@cps.msu.edu

ABSTRACT

Parallel processing is an area that is just beginning to be investigated in Genetic Programming (GP). To date a number of conflicting reports have been generated with respect to the effectiveness of parallel processing in GP. We report here that these conflicting reports may be due to the problem-dependent nature of parallel processing in GP which has not been found in GAs. This paper will review: what parallel processing and speed-up mean in the areas of GA/GP, some conflicting results that have been reported in the GP literature on whether parallel processing gives speed-up to GP problems, and offer an answer as to why different GP problems show different speed-up in parallel processing.

1.0 INTRODUCTION

Parallel processing is an issue that has often been examined in Genetic Algorithm (GA) research [Lin et. al. 1994, Manderick and Spiessens 1989, Mulhenbein 1989, Punch et. al. 1993, Tanese 1989, Pettey et. al. 1987]. It is worth noting that there are at least three senses of parallelism in the GA literature:

- *implicit parallelism*: This was a term first coined by Holland [Holland 1975] in some of the early descrpitions of GAs. Implicit parallelism is the ability of a GA to process approximately n^3 schemata for every n individuals evaluated in the population.
- *distributed parallel processing*: This is the decrease in processing time a GA requires to process the same number of indivudals as more processors are used. In the past, GAs have been described as *embarrassingly parallel*. By this, it is meant that a typical GA can easily be ported to run on multiple processors because of the simple processor communication required.
- *multiple population processing*: This is also a decrease in processing as found in the distributed case. However, this processing change is due to the fact that a single population of size *X* requires more evalutions to reach some level of performance than *n* subpopulations, each with *X*/*n* individuals if each subpopulation occasionaly exchanges solutions with other subpopulations. This change can be realized on either a single processor or multiple processors.

The distributed processing and multiple population cases can be conflated when multiple subpopulations are run on separate processors (see coarse-grain parallelism in Secion 2.0). Such a conflation leads to the claim that a parallel GA has *superlinear speedup*. Speedup is the measure of time decrease used by an algorithm as more processors are applied. Speedup is generally indicated as a relation, such as *linear speedup*. Linear speedup means that for every processor used, a linear decrease in time is observed in the run time of the algorithm. *Superlinear speedup* means that there is a larger than linear decrease in time for processor used. The use of the term superlinear speedup is problematic, especially in the area of parallel processing. Parallel processing researchers have claimed that superlinear speedup is

not possible in an algorithm where the only difference between single processor vs. multiple processor experiments is the number of processors. Summarized as Amdahl's law [Amdahl 1967], the argument is as follows. Suppose one records the time t that an algorithm takes to complete on a single processor. If, instead of being distributed across multiple processors, the algorithm is broken up into smaller segements (according to some parallelization approach) and those segments are run sequentially on the same single processor with t_{si} indicating the time needed for each segment. It is not possible that the sum time of all t_{si} be less than t since the same amount of work has to be done! The key point is the phrase "the same amount of work has to be done". We noted in the multiple population case that fewer evaluations **are** required. Clearly this means that less work is needed to reach the required level of performance. Thus GAs can achieve superlinear speedup, but under conditions that require less work be performed by the sum of all subprocessors than that needed by a single processor.

2.0 Parallel Processing in GAs

Parallelization is an area that has been much investigated in GAs for two reasons: GAs are very easy to parallelize and GAs typically require a lot of processing time to solve any real-world problem. As a result there are a number of approaches to parallelizing GAs, depending on the particular kind of problem being solved (see [Lin et. al. 1994] for more details):

- *micro-grain parallelism*: This is the simplest form of GA parallelism. Here, we essentially parallelize only the evaluation function. There is one master node which is responsible for all GA operations (selection, crossover, mutation etc.) *except* the evaluation function. When an indviduval must be evaluated, the master node exports that individual to a slave nodes for evaluation. When evaluation is completed, the fitness value is returned to the master node. When the master node has received back all the fitness values from each individual in the population, it continues with normal GA operation. Thus micro-grain parallelism is typically synchronous parallelism, since the master node must wait till all slave nodes have completed evaluation of their individuals.
- *fine-grain parallelism*: This form of parallelism is used to reduce problems of premature convergence by using a spatial distribution of individuals combined with a crossover operator based on locale. Each individual in the population occupies some location in a spatial distribution of the population. When these individuals crossover, they can only do so with their "near neighbors". Thus a particularly "good" individual can still dominate the population but the speed at which it does so is greatly reduced since it must go through some number of generations before its genetic material can be propagated to all other individuals in the space. This space of individuals can be divided such that processors are responsible for individuals located only in a particular part of the space (including having one individual per processor). Communication in this approach can require higher bandwidth depending on the topology of the space and the number of near neighbors since processors must pass individuals across boundaries for crossover. However, it is a less synchronous form of communication since it is not required that each processor proceed in lockstep with its neighbors.
- *coarse-grain parallelism (island parallelism):* This is an analogy to the kind of evolution seen by Darwin on the Gallopagos islands (hence it is also known as island parallelism). In this form of parallelism, the population is divided into autonomous subpopulations, where each subpopulation is completely controlled by a separate processor. Occassionally, each subpopulation exchanges some small number of individuals (exporting to other subpopulations, importing into itself). This results in an early, broad search of the search space by the multiple subpopulations, followed by a refinement of

where in the space each subpopulation searches based on what appear to be better answers found by neighbor subpopulations. This is also an asynchronous search process since processors (and subpopulations) do not have to be run in lockstep.

As mentioned in Section 1.0, the coarse-grain parallel case is interesting because it simultaneously uses two approaches that speed GA processing: the reduction in work obtained by having multiple subpopulations and the reduction in time obtained by having those subpopulations distributed across multiple processors.

2.1 Mutli-Population GAs

Since GAs are so easily parallelized, the most interesting aspect of parallel GAs is the reduction in work associated with multiple subpopulations. There are a number of factors which affect the amount of work reduced in multiple subpopulation GAs such as the number of subpopulations, the size of the subpopulations and others, but the three most important are the frequency of exchange, the number (and quality) of individuals exchanged, and the topology of near neighbors in the exchange. These have the most effect on the reduction in work/time.

We have conducted a number of experiments that explore these various parameters. For example, we evaluated the effectiveness of a number of different topologies and exchange approaches using a simple graph partitioning problem [Lin et. al 1994]. Here, we used two, 24 node 3-Cube-Connected-Cycle (3-CCC) graphs. The GA used an encoding of 48 bits, and searched for a partitioning of the 48 nodes into the two 24 node graphs. The initial parallel architectures explored were all ring-based architectures with a subpopulations running on its own separate processor. Of the 8 architectures evaluated, a number of conclusions could be made. First, as the number of subpopulations increase (that is, as the total population size was divided into smaller subpopulations, thus keeping the total number of individuals the same for all experiments) the number of optimal solutions increased. Second, as the number of subpopulations was increased, we observed a super linear speedup in solution time, indicating that not only was there a speedup from distributed processing, but also from a reduced workload due to mulitple populations. Third, some exchange topologies were more effective than others. For the 3-CCC problem, a *positive-distance* topology was most effective. Here, the topology of near neighbor was not fixed, but dynamic based on the Hamming distance of the best individuals in each subpopulation. Only those subpopulations whose best indivduals had a Hamming distance of less than 24 were allowed to make exchanges.

Finally, we began some experiments with a more radical topology termed the *injection architecture* topology. Rather than a ring topology for exchanges, the subpopulations were arranged in a hierarchy. This hierarchy had two interesting properties. First, as you traversed the hierarchy from the root to the leaf subpopulations, the representation used by those subpopulations was made coarser, less fine-grained. Thus the leaf subpopulations used a fairly coarse (abstract) representation of the problem, while it's parents used a more refined representation. Only the root node subpopulation used a full detailed representation. Second, the exchange of individuals was one-way, from coarse to fine grained subpopulations. The effect was to have coarse-representation subpopulations search a smaller, more abstract, space and then *inject* what appeared to be promising solutions into more fine grain representation subpopulations for more detailed examination. We continued our experiments using the CCC's, moving to four 3-CCC graphs requiring four partitions to be discoverd by the GA. In comparing the positive-distance, elitist, ring topology the injection architecture proved much more effective.

We further examined the effectiveness of the injection architecture topology on some more difficult, real-world problems. One such problem was the design of composite material beams, optimized to absorb energy for a fixed size [Punch et. al. 1995]. The beam was represented as a matrix of 24 layers of composite material, with each layer having 20 cells of material to be assigned in the layer. The GA represented the beam as a 480 element string, where each string element indicated a material to be assigned to some part of the matrix (the element size depended on the number of materials that could be used). As the evaluation function (a form of finite element analysis) of the beam was computationally expensive, we compared a micro-grained parallel approach to a simple ring topology multipopulation approach. As expected, while the micro-grained approach gave very nearly linear speedup, the ring topology gave more than linear speedup, again indicating a reduction in work due to the multiple subpopulations. Furthermore, we used an injection architecture topology where "coarseness" meant that submatrices of the beam were represented as a single element. Again, the injection architecture outperformed the ring architecture approach.

3.0 Genetic Programming

Our success with parallel GA architectures led us to wonder about the effectivness of parallelization approaches in genetic programming (GP) [Koza 1992]. Genetic programming is a close relative of GA. A GP consists of a set of individuals, a population. The individuals are evaluated by an external evaluation function, individuals are selected for various genetic operations (copying, mutation, crossover). There are, however, some important differences:

- the population inidviduals in a GA are strings while in GP strings are replaced by a tree representation.
- the trees in GP are of variable length.
- the trees in GP are typically (though not always) interpreted as programs, rather than solutions.
- because the individuals are programs, not solutions, each evaluation of the individual must be tested on many test cases, thus making GP even slower than GA approaches.

Since GP approaches share many of the same features as GAs, it seemed likely that the work done on parallelization of GAs would apply. While it is clear that distributed processing would work as in GAs, the question was whether multiple subpopulations would have the same effect.

3.1 Genetic Programming Parallelization

The first experiments we conducted were based on two problems, one a standard machine learning problem, the "ant path" problem, and a new problem we introduced which was inspired by Holland's royal road [Jones 1994], termed the "royal tree" problem.

The ant problem starts with a 32x32 matrix, where all of the matrix elements are initially empty. Some "path" through the matrix squares are filled with "food" for the ant to eat. The problem is
to start the ant at a standard point in the matrix, and given a set time period (typically something like 400 steps), see how many of the "food" particles that ant can pick up by walking over food elements.

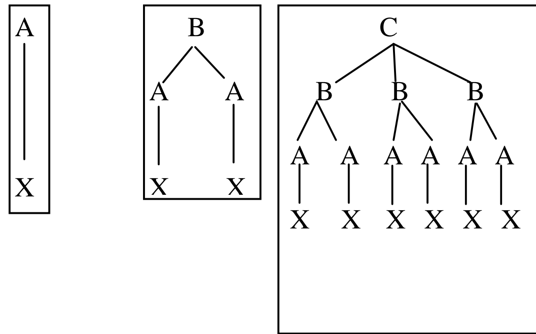

Figure 1: A perfect a-level, b-level and c-level royal tree

The royal tree [Punch et. al. 1996] is a problem that we developed to be used as a "standards" function for testing the effectiveness of GPs. It consists of a single base function that is specialized into as many cases as necessary, depending on the desired complexity of the resulting problem. We define a series of functions, *a*, *b*, *c*, etc. with increasing arity. (An *a* function has arity 1, a *b* has arity 2, and so on.) We also define a number of terminals *x*, *y*, and *z*. For any depth, we define a ``perfect'' tree as shown in Figure 1. A level-a tree is an *a* root node with a single *x* child. A level-b tree is a *b* root node with two level-a trees as children. A level-c tree is a *c* root node with three level-b trees as children, and so on. A level-e tree has depth 5 and 326 nodes, while a level-f tree has depth 6 and 1927 nodes.

The raw fitness of the tree (or any subtree) is the score of its root. Each function calculates its score by summing the weighted scores of its direct children. If the child is a perfect tree of the appropriate level (for instance, a complete level-c tree beneath a d node), then the score of that subtree, times a *FullBonus* weight, is added to the score of the root. If the child has the correct root but is not a perfect tree, then the weight is *PartialBonus*. If the child's root is incorrect, then the weight is *Penalty*. After scoring the root, if the function is itself the root of a perfect tree, the final sum is multiplied by CompleteBonus. Typical values used are: *FullBonus* = 2, *PartialBonus* =1 , *Penalty* = 1/3, and *CompleteBonus* =2. The score base case is a level-a tree, which has a score of 4 (the a---x connection is worth 1, times the FullBonus, times the CompleteBonus).

The reasoning behind this ``stair-step'' approach to the function is based on the reasoning originally used by the royal road. Many combinations of solutions can be found through genetic combination, but each *proper* combination gives a big jump in evaluation credit. The *FullBonus* is provided to give a large credit to those trees that find the correct, complete royal tree child. Since a deeper royal tree, such as a level-f tree, has a number of complete royal trees as children, each complete subtree found gives a large credit to that particular solution. The *PartialBonus* is used to give credit for finding the proper, direct child for a node, even if that direct child is not the root of a royal tree. This pressure is not as great as the *FullBonus*, but it is an effective incentive since the score is determined recursively down the tree and each node receives some credit when if finds its proper, direct children. If a node does not have the correct, direct children, it is penalized by *Penalty*, making the *FullBonus* and *PartialBonus* even more effective. Finally, if the resulting tree itself is complete, then a very large credit is given.

The reasoning behind the increase in arity required at each increased level of the royal tree is simple, we wanted to make a hard for GP to solve. That is, we could simply have required that a proper ordering of say a b-function (multiple levels of a 2-arity function) for the tree,

but requiring *increasing* arity as we climb to the next level dramatically increases the difficulty of the problem, and provides a measure of how well a GP can perform. For example, it is extremely difficult to climb to a level-f tree and we have never succeeded in climbing to level-g.

We ran three sets of experiments on a level-e royal tree and ant problem using a single population, a ring population and an injection architecture. These experiments were conducted using the lilgp GP programming system from the MSU GARAGe. Each experiment ran a maximum of 500 generations. The optimal value for the level-e royal tree is 122,880, and for the ant it was 89. Each problem was run 16 times. The total population size is 1000 for all experiments (1 subpopulation of 1000, or the sum of all subpopulations equal to 1000). The results of those experiments are shown in Table 1, Table 2 and Table 3. These results are reported as the number of *Wins* and *Losses*. The Wins are reported as *W:(x,y)* where *x* represents the number of optimal solutions found before 500 generations, and *y* is the average generation in which the the optimal solution was found. The Losses are reported as *L:(q,r,s),* where *q* is the number of losses (no optimal solution found before 500 generations), *r* is the average best-of-run fitness, and *s* is the average generation when the best-of-run occurred.

Table 1: Single Population Results

Problem	Over Selection (no mutation)	Prop. Selection (no mutation)	Over Selection (with mutation)	Prop. Selection (with mutation)
ant	W:(7,156) L:(9,78,198)	W:(2,265) L:(14,68,208)	W:(10,109) L:(6,73,300)	W:(7,112) L:(9,67,158)
royal tree	W:(1,145) L:(15,6144,47)	W:(0,0) L:(16,71,85)	W:(8,233) L:(8,9064,159)	W:(0,0) L:(16,71,92)

Table 2: Multi-population results (ring of 5 subpopulations)

Problem	Over Selection (no mutation)	Prop. Selection (no mutation)	Over Selection (with mutation)	Prop. Selection (with mutation)
ant	W:(4,160) L:(12,68,312)	W:(7,286) L:(9,71,257)	W:(6,208) L:(10,74,313)	W:(7,240) L:(9,73,181)
royal tree	W:(0,0) L:(16,10005,338)	W:(0,0) L:(16,83,62)	W:(0,0) L:(16,16284,373)	W:(0,0) L:(16,76,181)

Table 3: Injection architecture results, 4 nodes feeding into 1 final result node

Problem	Over Selection (no mutation)	Prop. Selection (no mutation)	Over Selection (with mutation)	Prop. Selection (with mutation)
ant	W:(2,297) L:(14,70,326)	W:(8,270) L:(8,70,272)	W:(2,116) L:(14,70,304)	W:(6,309) L:(10,74,256)
royal tree	W:(0,0) L:(16,20764,395)	W:(0,0) L:(16,81,152)	W:(0,0) L:(16,18354,405)	W:(0,0) L:(16,83,192)

These results are suprising given our previous results with parallelization and GAs. For the ant problem (except for the case of proportional selection with mutation) parallelization gave *poorer* results. These results are even more dramatically different for the royal tree problem, where no optimal result was *ever* found in 64 runs of various parallel processing approaches.

To confound things more, Koza & Andre [Andre and Koza 1996] reported at nearly the same time that, for the 5-parity problem they achieved super-linear speedup on a 64-node transputer, indicating that they got both the multi-population and distributed processing speedup.

3.1 Resolving the Differences in Parallel GP Results

We examined a number of issues to determine the cause of the observed discrepencies. For example, Andre & Koza exchange a much higher percentage of individuals than we did, but as expected this did not change the results. We tried other topologies and a host of different configurations to no avail. Finally we ran the same kind of experiment on a larger set of GP problems. We redid similar experiments on the regression problem. The regression problem is essentially a curve fitting problem. 20 points are proposed on the curve to be fit, and the GP attempts to generate a function that hits all 20 of the points on the curve to some small tolerance. We performed the experiment with some larger population sizes (4900), and more subpopulations (7). We did a single population and a ring of 7. Results for this experiment are shown in Table 4.

Table 4: Comparison of a single population of 4900 vs. a ring of 7x700 subpopulations for the regression problem

Architecture	Results	
Single Population (4900)	W:(6,46)	L:(9,16,181)
Ring of 7x700	W:(14,36)	L:(2,16,180)

Clearly the ring parallel processing did speedup the problem, as it did in the even-5 parity problem but unlike it did with the royal tree and ant problem.

We believe the reason has do with the "branching factor" associated with answers in the four observed problems. In both the regression and even-5 parity problem, only 2 argument functions were used. For the parity problem, the functions were {OR, NOT, NAND, AND} while in the the regression the functions were all 2 argument math functions {+, -, *, $/_{protected}$, sin, cos} For the other problems, functions using a higher number of arguments were allowed. For the ant, there were number of 3 and 4 argument functions {IF, PROG3, PROG4} and in the royal tree even a 5 argument function was allowed. Those problems with functions allowing larger number of arguments create trees with a higher number of nodes in a tree of fixed depth (like the ant and royal tree problems) than in problems whose functions have fewer arguments (like the regression and even-5 parity problem). In fact, even between the ant and royal tree problem, where the royal tree had higher branching than the ant, the royal tree performed more poorly than the ant.

We plan to confirm this more definitively with more tests of more GP problems, concentrating on the differences in branching factors between the problems.

4.0 CONCLUSIONS

In this paper we have reviewed a number of approaches to parallelism in GA and how these approaches work in GP. The effectiveness of parallel processing in GP appears to be problem-sensitive; those problems with high branching factors do not appear to benefit from parallel processing while those problems with low branching factors do. We will continue to explore this hypothesis by examining other GP problems and how the branching factor affects parallel performance.

REFERENCES

[Amdahl 1967] Amdahl, G. (1967), ``Validity of the Single processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings AFIPS Conference*, Vol. 30, pp.483-485, Thompson Books, Washington, D.C.

[Andre and Koza 1996] Andre, D. and J.R. Koza (1996) "Parallel Genetic Programming: A Scalabel Implementation Using the Transputer Network Architecture", *Advances in Genetic Programming 2*, editors Peter J. Angeline and Kenneth E. Kinnear, pp 317-338, MIT Press

[Holland 1975] J. Holland (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

[Jones 1994] Jones, T (1994). ``A Description of Holland's Royal Road Function,'' *Evolutionary Computation*, 2(4), pp. 409-415

[Koza 1992] Koza, J.R.(1992), *Genetic Programming*. Bradford/MIT Press.

[Lin et. al. 1994] Lin, S.-C., W. F. Punch, and E. D. Goodman (1994) ``Coarse-grain parallel genetic algorithms: Categorization and new approach.'' *Sixth IEEE SPDP*, pp 28--37, October.

[Manderick and Spiessens 1989] Manderick B., and P. Spiessens (1989), ``Fine-Grained Parallel Genetic Algorithms," *Third International Conference on Genetic Algorithms*, pp. 428-433, June.

[Mulhenbein 1989] Muhlenbein, H. (1989) ``ParallelGenetic Algorithms, Population Genetics and Combinatorial Optimization," *Third International Conference on Genetic Algorithms*, pp. 416-421, June.

[Pettey et. al 1987] C. Pettey, M. Leuze, and J. Grefenstette, (1987) "A Parallel Genetic Algorithm," *Proc. Second ICGA*, July, pp. 155-161.

[Punch et. at. 1995] Punch, W. F., R. C. Averill, E. D. Goodman, S.-C. Lin, and Y. Ding (1995)``Design Using Genetic Algorithms---some results for Composite Material Structures." *IEEE Expert*, 10(1), pp 42-49, February.

[Punch et. al. 1996] Punch, W.F., Doug Zongker and E. Goodman (1996), "The Royal Tree, a Benchmark for Single and Multiple Population Genetic Programming", *Advances in Genetic Programming 2*, editors Peter J. Angeline and Kenneth E. Kinnear, pp299-316, MIT Press

[Punch et. al. 1993] W. Punch, E. Goodman, M. Pei, L. Chai-Shun, P. Hovland and R. Enbody (1993), "Further Research on Feature Selection and Classification Using Genetic Algorithms," *Proc. Fifth ICGA*, June, pp. 557-564.

[Tanese 1989] R. Tanese (1989), "Distributed Genetic Algorithms," *Proc. Third ICGA*, June, pp. 434-440.