Genetic Programming on Clusters

Brian R Hanly and Daniel KL Tung

School of Computer Science and Software Engineering

Monash University

Clayton Campus, Melbourne, Australia

Email: hanly@cs.monash.edu.au

beetung@cs.monash.edu.au

Abstract

Genetic Programming is a relatively new field of artificial intelligence research that is finding a place in solving real world problems. One of the features of this method of problem solving is the high level of parallelism it exhibits. Thus the leading research institutes are using cluster computing as the platform on which to carry out their experiments.

Figure 1 – Evolution to the next Step[7]

1. Introduction

In this assignment we have attempted to report on the latest work done on Genetic Programming, especially when it has been executed on a Cluster Computing architecture. As we shall see though, the young lives of these two domains have had little chance to interact. It is felt that in a few more years when research into both areas is more mature, they will be happily married, as is their destiny.

1.1. Genetic Programming

Genetic Programming (GP) is a subset of Genetic Algorithms (GA) and Evolutionary Programming (EP) which are themselves subsets of the Artificial Life (AL) field of Artificial Intelligence (AI)[1]. AL is about modeling the behavior of biological units enabling them to function individually or collectively, improving over time, working towards a common goal. The catch phrase being "it worked for nature, so it can work for us too". AL is often good for finding an approximation to an answer fairly quickly, without having to search the whole problem space. AL methods are not guaranteed to converge to an optimal solution, but research show they often come closer faster than other methods of problem solving.

The GA is a model of machine learning which derives its behavior from a metaphor of the processes of evolution in nature[2]. This is done by the creation within a machine of a population of individuals represented by chromosomes, in essence a set of character strings that are analogous to the base-4 chromosomes that we see in our own DNA. The individuals in the population then go through a process of evolution. That is, GAs work on the idea that the solution to any problem is just a string of binary numbers, which is analogous to DNA in nature. When many solutions are found, they can be evaluated, giving each chromosome a score relative to the others so the good or 'fit' solutions can be made distinct from the bad. If we think of this in terms of natural selection, we can weed out unfit ones, or give them less chance of survival. When a population of chromosomes has been evaluated, it is time to replace it with the next generation. The next generation is found by applying reproduction and mutation operators to the old population, giving fitter chromosomes more opportunity to participate in these operations. A typical run might contain a population of one hundred chromosomes and go for one hundred generations. Note that one generation cannot start until the last is finished, but the evaluation of each chromosome is independent and so can be run in parallel. This is where parallel computer architectures such as clusters can help run experiments much faster.

Reproduction is achieved by taken the chromosomes of two members of the population and choosing one or more crossover points, thus mixing their genetic code. Reproduction is also known as the cross over operator and like in natural there is a chance you will get all the best genes from your mother and father, making you a better 'solution' than they were. Even though this can be thought of as sexual reproduction, there is no concept of male or female genes.

Mutation is simply flipping a couple of random bits in each chromosome, to see if that too results in a better chromosome for the next generation.

GP is a specialized form of GA. Consider that a computer program is the culmination of a series of operations on some data. We can say that the genes of the program are the operations it contains. This is the same as the chromosome made up of binary numbers as in GA, but the gene alphabet is not restricted to just 0 and 1. Also, the programs for each chromosome in a GP are best thought of in terms of their parse tree instead of a binary string.

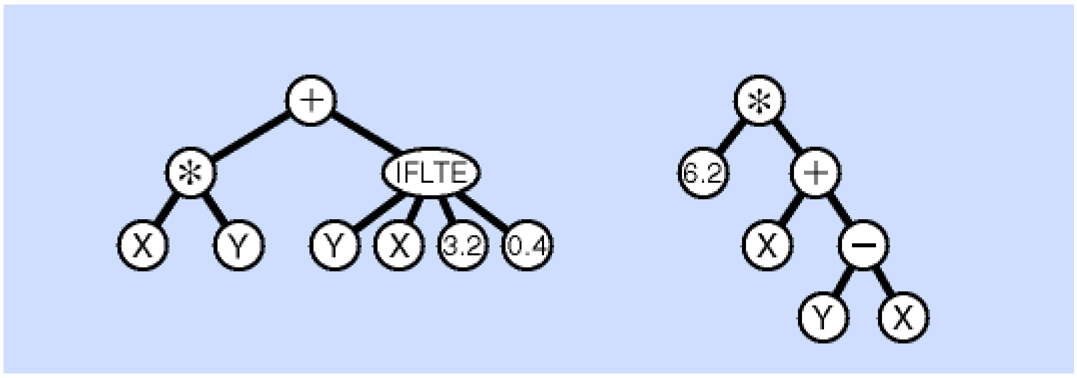

Figure 2 – Example of a GP Parse Tree.[3]

A GP run is started with a big population of thousands of randomly-generated computer programs. The set of functions that may appear can include ordinary arithmetic functions and conditional operators. The randomly created programs typically have different sizes and shapes. The best program in a particular run will evolve to have no unnecessary operations or loops. GPs have been used to evolve algorithms for problems that had taken people years of research to find traditionally. Note that these thousands of programs are completely independent of each other, and so each can be run on a separate cluster node.

The individuals in the initial random population and the offspring produced by each genetic operation are all syntactically valid executable programs. After many generations, a program may emerge that solves, or approximately solves, the problem as initially defined.

In the GP mutation operation we do more than just flip bits. A single parental program is probabilistically selected from the population based on fitness. A mutation point is randomly chosen, the sub-tree rooted at that point is deleted, and a new sub-tree is grown there using the same random growth process that was used to generate the initial population. Generating the initial population has been the focus of some interesting research[4]. Mutation is used less in GP than GA, say on less that 1% of genes, because it is likely to ruin a perfectly good program. It is used though, on the off-chance that it will make an improvement once in a while.

The crossover operator for GP is similar to that of GA. Two parental program chromosomes are probabilistically selected from the population based on fitness. The two parents participating in crossover are usually of different sizes and shapes. A crossover point is randomly chosen in the first parent and a crossover point is randomly chosen in the second parent. Then the sub-tree rooted at the crossover point of the first, or receiving, parent is deleted and replaced by the sub-tree from the second, or contributing, parent. Crossover is the predominant operation in GP and is performed on up to 90% of the chromosomes in the new population. That means ten percent of the new population is just a straight copy of chromosomes from the old population, with no operations performed on then at all.

GP has one more operation it can perform on its genes. Simple computer programs consist of one main program (called a result-producing branch). However, more complicated programs contain subroutines (also called automatically defined functions, ADFs, or function-defining branches), iterations (automatically defined iterations or ADIs), loops (automatically defined loops or ADLs), recursions (automatically defined recursions or ADRs), and memory of various dimensionality and size (automatically defined stores or ADSs). If a human user is trying to solve an engineering problem, he or she might choose to simply pre-specify a reasonable fixed architectural arrangement for all programs in the population (i.e., the number and types of branches and number of arguments that each branch possesses). GP can then be used to evolve the exact sequence of primitive work-performing steps in each branch.

However, sometimes the size and shape of the solution is the problem (or at least a major part of it). GP is capable of making all architectural decisions dynamically during the run. GP uses architecture-altering operations to automatically determine program architecture in a manner that parallels gene duplication in nature and the related operation of gene deletion in nature. Architecture-altering operations provide a way, dynamically during the run to add and delete subroutines and other types of branches to individual programs to add and delete arguments possessed by the subroutines and other types of branches. These architecture-altering operation quickly create an architecturally diverse population containing programs with different numbers of subroutines, arguments, iterations, loops, recursions, and memory and, also, different hierarchical arrangements of these elements. Programs with architectures that are well-suited to the problem at hand will tend to grow and prosper in the competitive evolutionary process, while programs with inadequate architectures will tend to wither away under the relentless selective pressure of the problem's fitness measure. Thus, the architecture-altering operations relieve the human user of the task of pre-specifying program architecture.

1.2 The Origins of Genetic Programming.

The fields of AL and EP has its earliest roots in a 1958 paper by R. M. Friedberg[5]. Friedberg saw that there was two ways in which computers could help people solve complex problems. They were by breaking tasks down so precisely that they could be coded into a computer, or by building a computer that does not need precise instructions. The former is the traditional approach and it works very well if there is a precise model of the task being considered. If there is a model, then an algorithm can be created. The latter approach is ideal for tasks where there is no well-defined model or formula, since we can't tell the computer exactly what to do anyway. The problem with the latter approach is that all computers require algorithms to operate. So Friedberg thought he could get around this dilemma by creating a program that did not work directly on the problem at hand, but spent its time creating another program. This other program could be tried on the problem, and refined until a solution was found.

There were several options open to Friedberg as to how he could get the main program to go about creating the little one, which he called Herman. One way to do it would be to randomly slap instructions together, and see if they did anything when they ran. However, if a program did do something useful, but only some of the time, it would still be thrown away the first time it failed. Ideally there would be a way of building on the slightly useful program, to eventually produce something that solved the problem for all inputs.

Friedberg decided that he would start by creating a very simple simulated computer that had 4 kinds of instructions, space for sixty-four of these 14 bit instructions and sixty-four bits of data. In the simulation, there was a Teacher, a Learner and Herman. The Teacher was the part of the simulation that worked out if the correct answer had been achieved yet, and the Learner was the part that made changes to Herman if the correct answer was not found. Herman was given input and the Teacher expected the correct output. Herman didn't know which bit or bits in the data area were input, nor into which bits he was expected to write the output, let alone what code he was supposed to contain to solve the problem. The Learner didn't know what the problem was either, it only knew that Herman had been bad and that he should change his ways.

Friedberg made the simulation this general, so that any input could be give to Herman, any output expected, and any mystery formula solved. This generality made the search space for the simulator very large indeed, but Friedberg was afraid of reducing the search space, as it would make the program less general, and might reduce the number of solutions available for each problem.

GA and GP claim their origins in this article because of the way Friedberg's Learner worked. Compared to a modern GP, Friedberg's Herman was a population of solutions of size two instead of thousands. As well as the Herman in memory, the Learner kept another copy of Herman, which was made up of completely different instructions. As time went by, the Learner would try swapping instructions from the secondary copy of Herman, depending on which of the two instructions was more often involved in the a successful simulation. This is analogous to the modern idea of cross over, but it was only done on one gene at a time. Also, at a certain time interval, the instruction with the lowest score in a certain group was replaced with a new, randomly generated instruction. This is similar to what we would call mutation today.

2. Works On Genetic Programming

There is a lot of research into GPs going on in the U.K., U.S., Japan, Europe and elsewhere. Researchers with enough funding have built or gained access to large cluster computing centers on which to run their experiments. The table below lists some the GP work we could find, whether or not it is actually being performed on a cluster architecture.

Our main source was the GP website[6]. It documents 15 books, 9 conferences, 1372 published papers, 37 completed PhDs, 58 underway, free software libraries for various platforms and languages, and 6 sets of course notes. This is a dot org site, which seems to be run largely by academics.

There are also two GP dot com websites, one with a hyphen[7] and one without[8].

Classify Works

Most of the work we have seen has been in academic research, with some people, who probably did PhDs and so-forth in the area, applying their skills and releasing commercial problem solving software that has GP at its heart. There are no commercial packages yet available that we could find built to run on clusters though. Its mostly just for stand alone MS Windows PCs and that kind of thing.

Name	Cluster Description	Remarks	URL
GP Inc Large Beowolf Style Cluster9	1000 Intel Pentium-II 350Mhz CPU cluster.	Success Story #1 below. The 1,000-Pentium machine was provided by Stan Fox of the COMPAQ Sunnyvale Staging Center. Most of the projects they have been running have been simulation of electronic circuits, circuit design, theoretical problem solving and some medical work. Their long term goal is to produce a system with human competitive intelligence.	http://www.genetic-programming.com/machine1000.html
GP Inc Medium Sized Beowolf Style Cluster[7]	70 Compaq (DEC) Alphas 533 MHz CPU Cluster.	Is used for smaller research projects, that don't need the full power of the 1000 node cluster.	http://www.genetic-programming.com
Evolutionary Computing & Intelligent Systems Research at University of Glasgow[10]	Parsytec SuperCluster of 64 T800 transputers running Parix and a PVM network of distributed UNIX workstations and high-grade PCs	Success Story #2 Below. This seems to be the main center of research for GP in the U.K.	http://www.elec.gla.ac.uk/groups/control/evonet/evonet_wg.html
AIM Learning Technology11	-	AIM Learning Systems is a company that has released a fully configurable commercial genetic programming package called DiscipulusTM. It is available for MS Windows PCs and is designed for help companies solve problems big and small. The professional version costs US$4000. AIM was invented in 1993 by Dr. Peter Nordin. It stands for Automated Induction of Machine Code and is a genetic programming environment that works to produce chromosomes made of Intel x86 instructions. Dr Nordin is so confident that he has taken out a patent on his idea. The web site does say anywhere that it runs on a cluster, only on	http://www.aimlearning.com/welcome.htm

		Windows PCs. I have no doubt that a future version of DiscipulusTM will be able to run on a company LAN, for example, using it overnight as a cluster. In the future many commercial applications will list this as a key feature.	
A Free GP Library12	-	This website is the home of a Genetic Programming library that has been put together over time but a variety of people. It is based in Austria and contains links to papers and work by people across Europe. The site includes free a C++ genetic programming class and some examples of its use. Again, none of these make explicit mention of whether they are using cluster computing. It has been tested on Linux.	http://aif.wu-wien.ac.at/%7Egeyers/archive/gpk/vuegpk.html
Grain Boundary Geometry Optimization Using a Genetic Algorithm[13]	Not specified, but appears in a list of successful cluster applications from 1995.	A Genetic Algorithm was combined with traditional molecular dynamics to simulate grain boundaries in metallic systems. Much more efficient that previous serial methods of solving this problem. So fast it allows real time interaction with the user. Was given the "Fastest Real Application" award in a super computer contest that year.	http://www.supercomp.org/sc95/proceedings/GII_HPC/HPC_APPS.HTM
DIY Clusters for Genetic Programming. [7]	10 DEC Alpha 533 MHz cluster	One of the pages at John Koza's GP.com site describes how you can build a Beowolf cluster of your own for as little as US$18000. That page appears to be nearly a year old, so its probably even cheaper now. They claim that such a cluster would be capable of about a half peta-flop (1015 floating-point operations) per day.	http://www.genetic-programming.com

3. Success Stories

Success Story #1: 1,000-Pentium Beowulf-Style Cluster Computer for Genetic Programming.9

Figure 3&4 – Two pictures of John's baby from their web-site.[7]

In July 1999, Genetic Programming Inc, headed by John Koza, started operating a new 1,000-node Beowulf-style parallel cluster computer consisting of 1,000 Pentium II 350 MHz processors and a host computer. The cluster has been designed specifically to run GP experiments and research. (I guess John has sold a bunch of books if he can build something like this.)

Basic Cluster Setup

This Beowulf-Style cluster system consists of 1,000 Pentium II 350-MHz processor units. It is constructed entirely from Commodity Off The Shelf (COTS) components. Each of the 1,000 Pentium II 350-MHz processors is resided in a grid pattern structure along with 64 megabytes of RAM each. There is no hard disk, video monitor, keyboard, or other input-output device associated with any of the nodes.

Communication between Cluster

A 100 megabit-per-second Ethernet Network Interface Card. Each processor communicates only with the server and four of its nearest neighbours in grid (North, East, South, West). This simplifies the architecture, and works well for running GP.

The 1,000 processors are logically arranged into a 25 by 40 toroidal grid. Each group of 40 processors that share a hub are arranged into one 8 by 5 rectangle within the larger grid.

Operation System

The processors run the GNU / Linux operating system (Red Hat Linux 6.0). This is the base operating system on every individual node. The Server computer is the same as the nodes but has more RAM and contains a 14 GB hard disk, a video display monitor, a floppy disk drive, a CD ROM drive, and a keyboard. They didn't just choose Linux because it is a free OS so they could spend more money on hardware. It was observed to be remarkably.

Running GPs

As previously stated, each processor communicates only with the server and four toroidally nearest neighbours (North, East, South, West). Except for occasional trace messages, the sole type of communication from the processors to the server during a run of genetic programming is the end-of-generation message. The end-of-generation message contains the best individual of the current generation (and certain additional statistics about the progress of the run). The sole type of inter-processor communication consists of the migration of a certain small number of individuals from one processor to each of its four toroidally adjacent processors. Thus each node is running its own GP job. The cluster is used to keep each run in touch with the others, by slowing spreading the knowledge learned throughout the system. Across the whole cluster the population of chromosomes might be anywhere between half to thirty million.

Approximately half of 64 MB of RAM associated with each of the 1,000 processors is available for the storage of the population. The remainder of the memory is used to house the operating system (shared by the two processors on a given motherboard), the GP application software, and various buffers used for exporting and importing individuals, and other items of overhead. All this RAM means there is no need for a hard disk at each node. Hard disks present potential reliability problems. The size of each population on a node is calculated before a run to make sure it will fit. During the run the population size can be changed anyway.

The server receives the end-of-generation reports from each processor. It creates an output file containing statistics about the run and the best results found so far. This output file is stored on the hard disk of the host computer

For most problems run on the cluster the largest proportion of time is taken by the evaluation of each chromosome to give it a fitness score. The other tasks, such as initialising the population at the start, find the highest score, performing cross-over and mutation, etc are all very quick. The design team have used these characteristics to get the highest possible level of efficiency from each node. That is, they are using the asynchronous island approach to parallelisation.

Broadly speaking, in the "island" approach the population for a given run is divided into semi-isolated subpopulations (called demes). Each sub-population is assigned to a node. The run begins with the one-time random creation of a separate population of individuals at each processor of the parallel computer system.

In the main generational loop of genetic programming, the task of measuring the fitness of each individual is first performed locally at each processor, as is the Darwinian selection and the genetic. The processors operate asynchronously in the sense that each generation starts and ends independently at each processor. Because each of these tasks is performed independently at each processor and because the processors are not synchronized, this asynchronous island approach to parallelisation efficiently uses all the processing power of each processor.

To spread the learned knowledge there has to be some communication between nodes. This id done by taking a relatively small percentage of the individuals, mostly the best ones – but each is given some chance to move, and copying it to the four neighbouring nodes. They are buffered until that node is ready to accept them into their population. The immigrants are typically inserted into the subpopulation at the destination processor in lieu of the just-departed emigrants of that processor's subpopulation.. Experience indicates that a modest amount of migration (say 2% of a processor's population, in each of four directions) is better than extremely high or extremely low amounts of migration. Thus, 8% of the individuals from a given processor migrate to neighboring processors on each generation. Not enough to stress the LAN I wouldn't think, especially since each migration is separated by substantially longer periods of time for fitness evaluation, etc. A generation in a run of parallel genetic programming may be of any length, but typically takes about 10 minutes to 1 hour. If a generation takes, say,10 minutes, about 150 generations can be run in a day. They are doing some serious computer over there at GP Inc.

Using these methods, nearly 100% efficiency is realized when genetic programming is run on a parallel computer system using the asynchronous island model for parallelisation. This near-100% efficiency is in marked contrast to the efficiency achieved in parallelising the most other kinds of computer calculations.

Fault Tolerance

The entire system is designed to be highly fault-tolerant. No processor is considered essential to the run. No acknowledgment is required for any message, including the messages containing emigrants from one processor to another. If a processor fails then there less emigrants to the surrounding nodes, but they don't worry about it, they just learn a little bit slower. This means even though the whole cluster is left running 24 hours a day, seven days a week, the staff can turn individual machines off for repairs, upgrades, etc without having to turnoff the whole system. When the machine is turned back on again it can join in the run with the rest. For GP, the new node will have a random startup population, which will usually be replaced fairly quickly by fitter chromosomes from the surrounding nodes after just a few generations. This is one of the most important features of cluster computing.

System Performance

A 1,000-node system generates a noticeable amount of heat and air conditioning (two 25-ton air conditioners) are required to keep the system at 70F degrees.

The 1,000-Pentium system operates at an aggregate clock rate of 0.350 Tera-Hertz. The SPECfp95 benchmark, which is a widely used and respected standard, rates the 350-MHz Pentium II processor at about 12. Thus the 1000 Node cluster delivers about 12,000 SPECft95. According to their figures, every CPU is running at 99% capacity on average, so the cluster really is running at 12,000. Good value.

Success Story #2: Evolutionary Computing & Intelligent Systems Research at University of Glasgow.[14]

Parsytec SuperCluster of 64 T800 transputers running Parix and a PVM network of distributed UNIX workstations and high-grade PCs.

This certainly sounds like an interesting combination of hardware. Clusters made up of different kinds of hardware must be very difficult to build and have operate reliably. This cluster has been made available to anyone performing research at the University. Much of that research has been in Evolutionary Computing, and some specifically on GP. There has been some work that uses GP to solve a problem and some refining GP techniques themselves, i.e. trying different variations of cross-over, mutation and reproduction to see if there are ways to make GP find better solutions faster. Pure algorithmic research such as this is vital for the maturity of the field, especially if it is accompanied by plenty of statistics from runs performed of the super cluster.

To quote one field of research completed recently, “Decimal coding with structural and parameter-range encoding capability in a hybridised GA/GP/EP/ES framework to systematically evolve the designs of sliding mode control systems, achieving globally optimised switching parameters, regions and laws/structures. Design automation techniques have also been developed for gain-scheduling, local controller network, neural network and block-diagram based nonlinear control systems, with both optimised coefficients and optimised structures.” Sounds pretty high-tech to me. It must be good.

Another researcher has been combining GP with another method of problem solving called Simulink. Perhaps it’s a product. “A powerful hybrid GP based technique has been developed and interfaced with Simulink building blocks. It is applied to evolve nonlinear engineering models from experimental data with added accuracy and sophistication and with optimised models utilising an 'unlimited range' of candidate structures. This work has also been applied to local model network based system identification.”

Unfortunately it was not possible to get any more information about these two projects. Judging from the other information at the site, the University has a keen interest in hardware design. Many of the projects they have undertaken have related to computer circuitry and hardware design, CAD, signal processing, pattern recognition, etc. Mathematically intense problems such as these are well suited to a cluster platform.

Success Story #3: GP written to run a robot soccer team and other short stories.[15]

Even though this article did not mention cluster computing at all, a team of five computers, in constant communication trying to solve a common problem fits the description. The difference here is that the computers are onboard mobile robots whose problem is trying to score goals against another robot team.

The Robo Cup team of Carnegie Mellon University, headed by David Andre and Astro Teller play by following strategies evolved using GP. This is a reasonable approach for a hard problem like controlling the motion of a ball. It would be very hard to manually program the robots to have a rule for every situation it might come up against on the soccer field.

Andre and Teller's robots came in 17th out of 36 teams. A good effort considering that the competing software had been written by some of today's best artificial intelligence researchers. The Darwin United team is "not bad for something that was created out of thin air," comments John Koza, a consulting professor of medical informatics at Stanford University, who is considered (or considers himself) the inventor of genetic programming.

The article goes on to point out that programs produced by GP are often convoluted, bizarrely multi-layered creations, nothing like the software a human might write. But they are remarkably powerful and flexible.

Genetic programming can be used to tackle a wide variety of problems, such as creating satellite controllers, designing artificial limbs and solving tricky problems in molecular biology. The beauty of genetic programming is that the programmer doesn't need to be an expert in satellite design, physiology or molecular biology to do this. The programmer creates a "fitness measure" -- a way for the software to know when it has successfully "evolved" a working solution -- and the software arrives at the solution using the evolutionary process of GP.

Early work in GP focused on designing circuits similar to those found in stereos. The "problem" given to the software was to design a type of circuit known as a low-pass filter; a successful "solution" would be a circuit that could pass low-frequency sounds to the stereo's woofer. The "fitness measure" in that case was the ability to separate the low- and high-frequency sounds. This work is described in John Koza's latest book, Genetic Programming III[16]. In one chapter it describes a flaw they found in SPICE, which is a widely used electric circuit computer simulator package. I think the Digital Systems department uses it at Clayton. They used SPICE to help in the evaluation of the GP population when they were trying to evolve an amplifier. Some how the GP generated programs had found a flaw in SPICE, which no-one had ever noticed before. It was able to use small voltages to produce impossible levels of signal amplification. This had Koza confounded, and in the end he had to change the GP to automatically kill any member of the GP population that was able to perform better than perfect amplification.

The article goes on to say that in 1996, Koza used genetic programming to evolve a design for a specific type of electrical circuit. The computer produced several solutions; but several actually infringed on 21 patents filed earlier this century. This shows that GP results can closely match ideas conceived by humans.

The article then talks about a number of GP projects, none of which explicitly mention that they were executed on a cluster platform, though any of them could have been. Some of the projects mentioned include data structure evaluation, protein-analysis, the creation of art and music,

On a very complex mathematical problem called the 1-d majority cellular automata problem, a GP-produced solution was the best in the world for about a year. It was eventually beaten out by a solution created by another evolutionary algorithm similar to GP. The human-created solution to this problem remains in third place.

8. Summary and Conclusions

The field of Genetic Programming has really only been around since the early nineties, despite Friedberg's work in 1958. Certainly, five years ago hardly anyone was researching the field, and many in the Genetic Algorithms domain dismissed GP (which is how they had been treated by other segments of the AI community). Now dozens of the best universities research the area and actively teach it to their students. Cluster computing is also a relatively new field, with standards such as MPI still being developed and made more popular among academics and private research groups. All the GP literature makes point of its inherent parallelism but few go as far as doing it. It is our opinion that in another five years, Genetic Programming on Clusters will be very widespread, and people will wonder how that ever did without it.

References

[1] ftp://ftp.cerias.purdue.edu/pub/doc/EC/Welcome.html

[2] http://www.genetic-algorithms.org

[3] http://www.genetic-programming.com/gpanimatedtutorial.html

[4] Böhm W., Geyer-Schulz A., "Exact Uniform Initialization for Genetic Programming", Foundations of Genetic Algorithms 4, (Hrsg. Richard K. Belew und Michael Vose), Morgan Kaufmann, San Francisco, 379-407, 1997.

[5] "A Learning Machine: Part I", R. M. Friedberg, IBM Journal, January 1958.

[6] http://www.genetic-programming.org

[7] http://www.genetic-programming.com

[8] http://www.geneticprogramming.com

[9] http://www.genetic-programming.com/machine1000.html

[10] http://www.elec.gla.ac.uk/groups/control/evonet/evonet_wg.html

[11] http://www.aimlearning.com/welcome.htm

[12] http://aif.wu-wien.ac.at/%7Egeyers/archive/gpk/vuegpk.html

[13] http://www.supercomp.org/sc95/proceedings/GII_HPC/HPC_APPS.HTM

[14] http://www.elec.gla.ac.uk/groups/control/evonet/evonet_wg.html

[15] http://www.genetic-programming.com/published/Salon081099.html

[16] Genetic Programming III – Darwinian Invention and Problem Solving, John R Koza, et al, Kaufman, 1999.