PARALLEL GENETIC PROGRAMMING ON A NETWORK OF TRANSPUTERS

John R. Koza
Computer Science Department
Stanford University
Stanford, CA 94305-2140 USA
Koza@CS.Stanford.Edu
Phone: 415-941-0336

David Andre
Visiting Scholar
Stanford University
andre@flamingo.stanford.edu
Phone: 415-326-5113

ABSTRACT

This report describes the parallel implementation of genetic programming in the C programming language using a PC 486 type computer (running Windows) acting as a host and a network of transputers acting as processing nodes. Using this approach, researchers of genetic algorithms and genetic programming can acquire computing power that is intermediate between the power of currently available workstations and that of supercomputers at a cost that is intermediate between the two.

A comparison is made of the computational effort required to solve the problem of symbolic regression of the Boolean even-5-parity function with different migration rates. Genetic programming required the least computational effort with an 8% migration rate. Moreover, this computational effort was less than that required for solving the problem with a serial computer and a panmictic population of the same size. That is, apart from the nearly linear speed-up in executing a fixed amount of code inherent in the parallel implementation of genetic programming, parallelization delivered *more than linear* speed-up in solving the problem using genetic programming.

1. Introduction

Amenability to parallelization is an appealing feature of genetic algorithms, evolutionary programming, evolution strategies, classifier systems, and genetic programming (Holland 1975, Tanese 1989, Goldberg 1989a, Stender 1993).

The probability of success in applying the genetic algorithm to a particular problem often depends on the adequacy of the size of the population in relation to the difficulty of the problem. Although many moderately sized problems can be solved by the genetic algorithm using currently available workstations, more substantial problems usually require larger population sizes. Since the computational burden of the genetic algorithm is directly proportional to the population, more computing power is required to solve more substantial problems.

Increases in computing power can be realized by either increasing the speed of computation or by parallelizing the application. Fast serial supercomputers and current parallel supercomputers are,

of course, expensive and may not be accessible. Therefore, we focus our attention on parallelization.

Section 2 of this report describes our search for a practical option that provides computing power that is intermediate between that of workstations and supercomputers at a price (in terms of both money and programming effort) that is intermediate between that of workstations and supercomputers.

Section 3 then describes the successful parallel implementation of genetic programming in the C programming language using a PC 486 type computer (running Windows) and a network of transputers.

Section 4 compares the computational effort required to solve a problem using the parallel computer system with semi-isolated subpopulations and a serial computer using panmictic selection (i.e., where the individual to be selected to participate in a genetic operation can be from anywhere in the population).

Section 5 is the conclusion.

2. Selection of Machinery to Implement Parallel Genetic Programming

This section describes our search for a practical method for implementing parallel genetic programming.

2.1 Time and Memory Demands of Genetic Algorithms

The genetic operations (such as reproduction, crossover, and mutation) employed in the genetic algorithm and other methods of evolutionary computation can be rapidly performed on a computer and do not consume large amounts of computer time. Thus, for almost any practical problem, the dominant consideration as to computer time for the genetic algorithm is the evaluation of the fitness (whether explicit or implicit) of each individual in the population.

Fitness measurement is time-consuming for many reasons, including the time required to decode and interpret each individual in the population, the necessity of computing fitness over numerous separate fitness (environmental) cases, the necessity of conducting lengthy simulations involving numerous incremental time steps (or other finite elements), the time required to make complicated state transition calculations for the simulations, the time-consuming nature of the primitive functions of the problem, and the time required to compute reasonably accurate averages of performance over different initial conditions or different probabilistic scenarios. Fortunately, these time-consuming fitness evaluations can be performed independently for each individual in the population.

The dominant consideration as to memory is the storage of the population of individuals (since the population typically involves large numbers of individuals for non-trivial problems). Storage of the population does not constitute a major demand as to computer memory for the genetic algorithm operating on fixed-length character strings; however, memory usage is an important consideration when the individuals in the population are large program trees of varying sizes and shapes (as is the case in genetic programming).

2.2 The Island Model of Parallelization

There are several distinct ways to parallelize genetic algorithms, including parallelization on the basis of individuals in the population, fitness cases, or independent runs (Goldberg 1989a, 1989b; Koza 1992). Parallelization on the basis of individuals in the population is discussed below.

In a fine-grained parallel computer (e.g., the MasPar machines or the CM-2 with 65,536 processors), each individual in the population can be mapped onto one processor (as described in Robertson 1987 for genetic classifier systems).

In a coarse-grained or medium-grained parallel computer, larger subpopulations can be situated at the separate processing nodes. This approach is called the *distributed genetic algorithm* (Tanese 1989) or the *island model* for parallelization. The subpopulations are often called *demes* (after Wright 1943).

When a run begins, each processing node locally creates its own initial random subpopulation. Each processing node then measures the fitness of all of the individuals in its local subpopulation. Individuals from the local subpopulation are then probabilistically selected based on fitness to participate in genetic operations (such as reproduction, crossover, and perhaps mutation) based on their fitness. The offspring resulting from the genetic operations are then measured for fitness and this iterative process continues. Because the time-consuming measurement of fitness is performed independently at each separate processing node, this approach to parallelization delivers an overall increase in performance that is nearly linear with the number of independent processing nodes (Tanese 1989).

Upon completion of a generation involving a certain designated number of genetic operations, a certain number of the individuals in each subpopulation are probabilistically selected for emigration to a designated (e.g., adjacent) node within the topology (e.g., toroidal) of processing nodes. When the immigrants arrive at their destination, a like number of individuals are selected for deletion from the destination processor. The interprocessor communication requirements of these

migrations are low because only a small number of individuals migrate from one subpopulation to another and because the migration occurs only after completion of the time-consuming fitness evaluation.

There are, of course, numerous variations in this overall approach. For example, the selection of emigrants may be based on fitness or at random using a uniform probability distribution. The selection of individuals to be deleted may be based on unfitness or at random (or the individuals to be deleted may simply replace that node's recently departed emigrants). In the so-called *steady-state* approach, the generation consists of only one genetic operation; however, in the so-called *generational* approach, the generation consists of a large number of genetic operations (typically of the order of the population size). The generations may be synchronized or not. The amount of migration may be small (thereby potentially creating distinct subspecies within the overall population) or large (thereby approximating, for all practical purposes, a panmictic population).

When any individual in any subpopulation satisfies the success predicate of the problem, this fact is reported to the supervisory process that controls all the processing nodes in the parallel system; the entire process is then stopped; and the satisfactory individual is designated as the result of the overall run. Runs may also stop when a specified targeted maximum number of generations have been run.

The distributed genetic algorithm is well suited to loosely coupled, low bandwidth, parallel computation.

2.3 Design Considerations for Parallel Genetic Programming

The largest of the programs evolved in *Genetic Programming* (Koza 1992) and *Genetic Programming II* (Koza 1994a) contained only a few hundred points (i.e., the number of functions and terminals actually appearing in the work-performing bodies of the various branches of the overall program).

The system for the parallel implementation of genetic programming described herein was specifically designed for problems requiring evolved programs that are considerably larger than those described in the two books and whose fitness evaluations are considerably more time-consuming than those described in the two books. For design purposes, we hypothesized multi-branch programs that would contain at most 2,500 points.

Multi-branch computer programs of the type that are evolved with genetic programming can usually be conveniently and economically stored in memory using only one byte of storage per point. In this representation, each point represents either a terminal or a function (with a known arity). One byte of storage can represent 256 possibilities. One byte can therefore accommodate several dozen functions (i.e., both primitive functions and automatically defined functions), several dozen terminals (i.e., actual variables of the problem and zero-argument functions), while still leaving room for around 150 to 200 different random constants (if the problem happens to employ random constants). Thus, a population of 1,000 2,500-point programs can be accommodated in 2.5 megabytes of storage with this one-byte representation. In the event that the population is architecturally diverse (as described in Koza 1994a, 1994b), the actual memory requirements approach 3 megabytes.

We further hypothesized, for purposes of design, a measurement of fitness requiring one second of computer time for each 2,500-point individual in the population. With this assumption, the evaluation of 1,000 such individuals would require about 15 minutes and 50 such generations could be run in about a half day, 100 such generations would occupy about a day, and 200 such generations would occupy two days.

Since problems obviously vary considerably as to the size of the evolved programs in the population, the amount of time required to measure fitness, the population size required to efficiently solve the problem, and the number of generations per run, we also considered variants of the above design criteria (e.g., 5,000 500-point programs) including smaller programs and various shorter and longer times for the measurement of fitness of a single individual of the population.

Thus, our design criteria were centered on a parallel system with

• a one-second fitness evaluation per program,
• 1,000 2,500-point programs per processing node, and
• one-day runs consisting of 100 generations.

Our selection criteria was the overall price-to-performance ratio.

"Performance" was measured (or estimated) for the system executing our application involving genetic programming, not generic benchmarks (although the results were usually similar).

"Price" includes not only the obvious costs of the computing machinery and vendor-supplied software, but also includes our estimate of the likely cost, in both time and money, required for the initial implementation of the approach, the ongoing management and maintenance of the system, and the programming of a succession of new problems in the area of genetic programming. The price of initial implementation, of course, depends on the overlap between the expertise required to implement a particular approach and our own particular existing capabilities. The price of ongoing management and maintenance included our estimate of how much time and money would be expended in keeping the hardware and software of the system operational. In other words, our selection criteria reflected our desire to actually do research in the area of genetic programming (and not to do research in parallel computation, not to do research in electrical engineering, not to become enmeshed in major problems of system management and maintenance, and not to expend significant effort in programming each new problem).

Using the above design and selection criteria and our price-to-performance ratio as a guide, we examined serial supercomputers, parallel supercomputers, networks of single-board computers, workstation clusters, special-purpose microprocessors, and transputers.

Since we had previously been using a serial computer, a extremely fast, well priced serial supercomputer would optimally satisfy our criteria as to the cost of initial implementation, ongoing management and maintenance, and programming of new problems. Current extremely fast serial supercomputers (e.g., Cray machines) attain their very high speeds by vectorization or pipelining. We believed (and Min 1994 verified) that these machines would not prove to deliver particularly good performance on genetic programming applications because of the disorderly sequence of conditional and other operations in the program trees of the individuals in the population. Moreover, fast serial supercomputers are very expensive and there is an issue of affordability.

Although a serial solution would have been ideal, we believe that greater gains in computing power for doing experimental work on large problems in the field of genetic programming will come, in the next decade or two, from parallelization (rather than increased computer speed) and that parallelization is going to prove to be a necessary step to take in this field.

The fine-grained SIMD ("Single Instruction, Multiple Data") supercomputers (such as Thinking Machine's CM-2 and MasPar's machines) are inappropriate for genetic programming since the individual programs in the population are generally of different sizes and shapes and contain numerous conditional branches. While most of the comments in this report apply to both genetic programming and genetic algorithms operating on fixed-length character strings, note that a fine-grained SIMD machine may be very suitable for applications of the genetic algorithm since this objection may not apply to the genetic algorithm operating on fixed-length character strings. It should also be noted that it may, in fact, be possible to efficiently use a fine-grained SIMD machine for an application (such as genetic programming) that seemingly requires a MIMD machine (Dietz and Cohen 1992, Dietz 1992). However, successful practical implementation of this approach appeared to require substantial expertise in areas outside our existing capabilities. It is not clear whether this concept would deliver any net benefit on any existing machine for programs containing a large number of primitive functions. Moreover, each new problem undertaken in genetic programming would require extensive customizing of the details of parallelization so that there would be high ongoing price of programming new problems in such a system.

The coarse-grained and medium-grained MIMD ("Multiple Instruction, Multiple Data") supercomputers (such as Thinking Machine's CM-5, Intel's Paragon, the NCUBE machines, and the Kendall Research machines) do not appear to be a cost-effective way to do genetic programming because they are usually designed to deliver a large bandwidth of inter-processor

communication at the expense of computational power. The price of each processing node of a typical supercomputer was in the low to middle five-figure range while the microprocessor at each node was usually equivalent to that of a single ordinary workstation. In some cases (e.g., Thinking Machine's CM-5), processing could be accelerated by additional SIMD capability at each node, but most genetic programming applications cannot take advantage of this particular capability. The fact is that when one buys any of these coarse-grained and medium-grained MIMD machines, one is primarily buying high bandwidth inter-processor communication, rather than raw computational power.

We also considered the idea of networks of workstations or networks of single-board computers (each containing an on-board Ethernet adapter); however, these approaches present formidable practical issues involving the ongoing management and maintenance of the independent processes on an unruly collection of machines. Although it is possible to use independent computers in this manner (Singleton 1993), these computers (and most of the supporting software) were designed for independent operation of the machines.

Printer controller boards are inexpensive, have substantial on-board memory, have on-board Ethernet connectivity, and have very high processing capabilities (often being more powerful than the computer they serve). However, these devices are typically programmed once (at considerable effort) for an unchanging application and there is insufficient high-level software for writing and debugging new software for a constant stream of new problems.

Solutions involving the Intel i860 microprocessor were avoided because the disorderly sequence of conditional and other operations in the program trees in genetic programming interferes with the pipelining that gives this device its high performance.

We will not recount all the details of the process by which various alternatives were dismissed for our particular application involving genetic programming. We now turn to a detailed description of the approach that was eventually chosen.

2.4 Transputers

A transputer is a single VLSI device containing an on-chip processor, on-chip memory, and several independent serial bi-directional physical on-chip communication links.

Manufactured by INMOS (a division of SGS-Thomson Microelectronics), the transputer was the first microprocessor that was specifically designed to support multiprocessing. The transputer was also one of the first 32-bit microprocessors. Transputers are designed to operate in parallel and their on-chip communication links and multiprocessing capabilities were specifically designed to facilitate parallel supercomputing.

Transputers are typically mounted on small boards, called TRAMs, with up to 16 megabytes of additional RAM memory above and beyond the on-chip memory. One currently popular model of TRAM (occupying approximately the volume of a pile of a half dozen credit cards) has four megabytes of RAM memory and an INMOS T-805 transputer featuring a 30 MHz 32-bit floating-point processor, four kilobytes of on-chip memory, and four communication links. TRAMs and other transputer-based products are manufactured by companies such as Transtech of Ithaca, New York (our source); Microway of Kingston, Massachusetts; Alta of Sandy, Utah; Parsytec of West Chicago, Illinois; Alex of Montreal; and others.

Expansion boards housing a number of TRAMs are available for PC types of computers and Sun workstations. For example, the INMOS B008 board is a standard PC type of expansion board that houses up to 10 INMOS T-805 4-megabyte TRAMs. Additional TRAMs may be housed on INMOS B014 motherboards mounted in a VME box, on additional B008 expansion boards within the PC computer (if it has adequate power supply and ventilation), or on additional B008 expansion boards in a PC expansion box.

One important reason for considering the transputer was that it was designed to support parallel computation involving multiple inter-communicating processes on each processor and inter-processor communication. The transputer was designed so that processes could be easily loaded onto all the processing nodes of a network, so that the processes on each node can be easily started up, so that messages can be easily sent and received between two processes on the same or

different processing nodes, so that messages can be easily sent and received between one processing node and a central supervisory program, so that messages can be easily sent and received by host computer from the central supervisory program, and so that the processes on each node could be easily stopped. Moreover, debugging tools exist.

When the transputer was introduced in 1985, the transputer's 32-bit architecture, its multiprocessing capabilities, and it orientation to parallel supercomputing were ahead of its time. Moreover, the transputer was originally conceived and presented as "an OCCAM machine." This strong association with a nonstandard proprietary programming language caused the transputer to encounter considerable resistance to widespread adoption (a history similar to that of other specialized machines, such as the LISP machines). Today, mainstream programming languages (notably C) are usually used for programming transputers and mainstream computers (such as PC type computers and Sun workstations) are usually used as the front-end computers for networks of transputers. Curiously, although several hundred parallel supercomputers containing 100 to 1,000 and more transputers were assembled, the transputer achieved commercial viability and success as embedded processors involving only a single transputer (because of their multiprocessing and communication capabilities and the fact that an entire small computer was on a single chip).

As it happens, there are a considerable number of successful applications of networks of transputers in the parallel implementation of genetic algorithms operating on fixed length character strings (Abramson, Mills, and Perkins 1994, Bianchini and Brown 1993, Cui and Fogarty 1992, Kroger, Schwenderling, and Vornberger 1992, Schwehm 1992, Tout, Ribeiro-Filho, Mignot, and Idlebi 1994; Juric, Potter and Plaksin 1995).

2.5 Suitability of a Transputers for Parallel Genetic Programming

The transputer was then considered in detail in light of its interprocessor communication capability, speed, memory, and cost.

The communication capabilities of the transputer are more than sufficient for implementing the island model of genetic programming. The bi-directional communication capacity of the INMOS T-805 transputer is 20 megabits per second simultaneously in all four directions. If 1% of the individuals at each processing node are selected as emigrants in each of four directions, each of these four boatloads of emigrants would contain 10 2,500-byte individuals (or 50 500-byte individuals). If 8% of the individuals at each processing node are selected as emigrants in each of four directions, each of these four boatloads of emigrants would contain 80 2,500-byte individuals (or 400 500-byte individuals). The communication capability of 20 megabits per second is obviously more than sufficient to handle 25,000 or 200,000 bytes every 15 minutes.

Our tests indicate that a single INMOS T-805 30-MHz microprocessor is approximately equivalent to an Intel 486/33 microprocessor for a run of genetic programming written in the C programming language. The transputer is about half as fast as an Intel 486/66 machine. Although a single 486/33 is obviously currently not the fastest microprocessor, this microprocessor is capable of doing a considerable amount of computation in one second. One second of 486/33 computation per fitness evaluation seems to be a good match for the general category of problems that we were envisioning.

As previously mentioned, 1,000 2,500-point programs (or 5,000 500-point programs) can be stored in 2.5 megabytes of memory (about 3.0 megabytes, if the population happens to be architecturally diverse). On a 4-megabyte TRAM, there is, therefore, about 1 to 1.5 megabytes of memory remains after accommodating the population. This remaining amount of memory is sufficient for storing the program for genetic programming, the communication buffers, the stack, and other purposes while leaving some space remaining for possible problem-specific uses (e.g., special databases of fitness cases). Thus, a TRAM with the INMOS T-805 30-MHz processor with 4 megabytes of RAM memory satisfies our requirements for storage. Note that we did not select TRAMs with 8 or 16 megabytes because the one processor would then have to service a much larger subpopulation (thereby increasing the time required for the fitness evaluation of the entire population).

TRAMs cost considerably less than $1,000 each in quantity. Thus, the cost of a parallel system capable of handling a population of 64,000 2,500-point programs (or 320,000 500-point programs) with the computational power of 64 486/33 processors can be acquired for a total cost that is intermediate between the cost of a single fast workstation and the cost of a mini-supercomputer or a supercomputer. Moreover, a system with slightly greater or lesser capability could be acquired for a slightly greater or less cost.

Moreover, the likely amount of time and money required for initial implementation, ongoing management and maintenance, and programming of new problems seemed to be (and has proved to be) low.

3. Implementation of Parallel Genetic Programming Using A Network of Transputers

This section describes our implementation of parallel genetic programming in the C programming language using a PC 486 type computer running Windows as a host and a network of INMOS transputers.

3.1 The Physical System

A network of 66 TRAMs and one PC 486/66 486 type computer are arranged in the overall system as follows:

(1) the host computer consisting of a keyboard, a video display monitor, and a large disk memory,

(2) a transputer running the debugger,

(3) a transputer containing the central supervisory process, (the Boss process), and

(4) the 64 transputers of the network, each running its own Monitor process, a Breeder process, an Exporter process, and Importer Process.

The PC computer is the host and acts as the file server for the overall system. Two TRAMs are physically housed on a B008 expansion board within the host computer. The first TRAM is dedicated to the INMOS debugger. The second TRAM (called the "Boss Node") contains the central supervisory process (called the "Boss") for running genetic programming. The Host computer is physically linked to the transputer containing the Debugger process (the Debugger node). The Debugger node is, in turn, physically linked to the transputer containing the Boss process (the Boss node).

The remaining 64 TRAMs (the processing nodes) are physically housed on 8 boards in two VME boxes. The 64 processing nodes are arranged in a toroidal network in which 62 of the 64 processing nodes are physically connected to four neighbors (in the N, E, W, and S directions) in the network. Two of the 64 nodes of the network are exceptional in that they are physically connected to the Boss Node and only three neighbors. The Boss Node is physically linked to only two transputers of the network.

The communication between processes on different transputers is by means of one-way, point-to-point, unbuffered, synchronized channels. The channels are laid out along physical links either by the programmer or using a virtual router provided by the manufacturer.

Figure 1 shows the various elements of the overall system for implementing the island model of genetic programming. The boxes in the figure denote the various computers, including the host computer (a PC 486 type machine), the Debugger Node (a transputer), the Boss Node (a transputer), and the network of processing nodes (transputers). For simplicity, the figure shows only nine of the 64 processing nodes. The ovals denote the various files (all on the host computer), including the one input file for each run (containing the parameters for controlling the run) and two output files (one containing the intermediate and final results of the run and one containing summary information needed by a monitoring process written in Visual Basic). The rounded boxes denote the input-output devices of the host computer, including its keyboard and video display monitor. The diamond denotes the Visual Basic monitoring process that displays information about the run on the video display monitor of the host computer. Heavy unbroken lines are used to show the physical linkage between the various elements of the system. Heavy broken lines show the lines of virtual communication between the Boss node and the processing nodes. Light unbroken lines show the lines of virtual communication connecting each of the processing nodes with their four neighbors.

The host 486 computer uses Windows as its operating system. No operating system is required for the transputers since they do not access files or input-output devices and all priority scheduling and communications are handled by the hardware of the transputer itself.

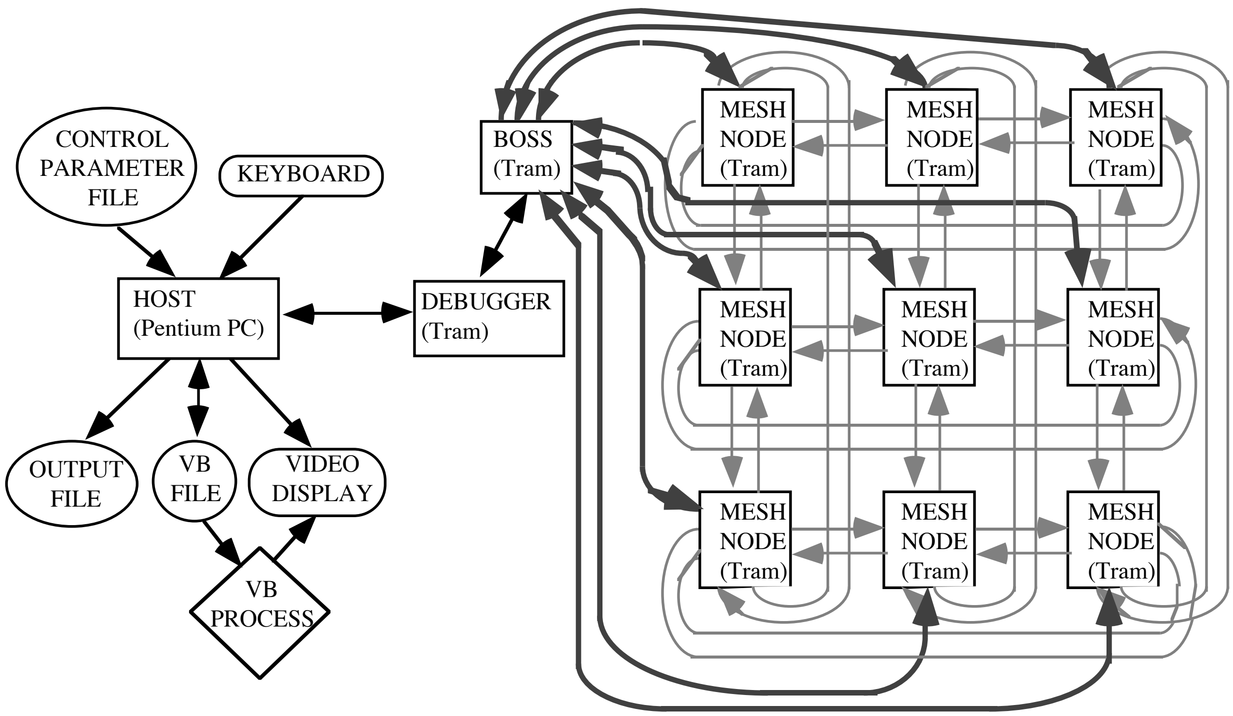

Figure 1 Overview of the parallel system.

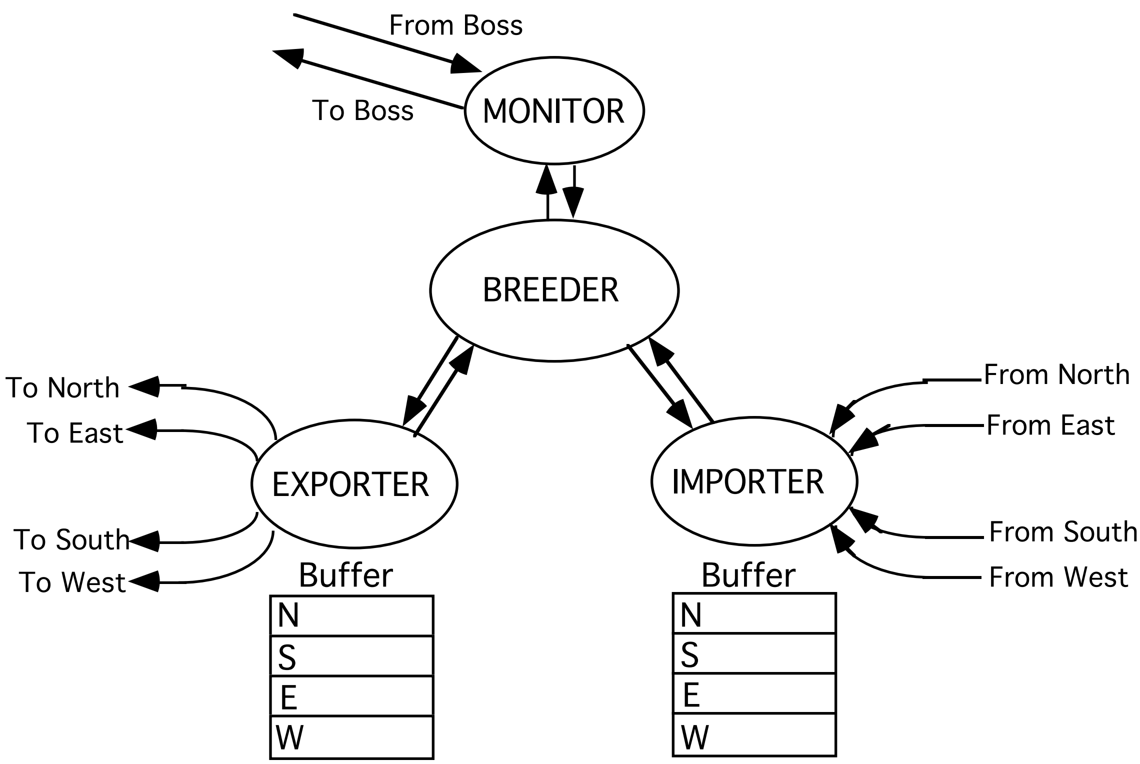

Figure 2 The four processes on each of the 64 processing nodes.

3.2 The Inter-Communicating Processes

Transputers are programmed using inter-communicating processes connected by channels.

The Host computer runs two processes. The Host process on the Host computer receives input from the keyboard, reads an input file containing the parameters for controlling the run, writes the two output files, and communicates with the remainder of the system. The second process on the Host computer is the Visual Basic monitoring process.

The TRAM with the Debugger runs only one process, the INMOS-supplied Debugger process.

The TRAM with the Boss runs only one process, the Boss process.

Each of the 64 TRAMs in the network concurrently runs the following four processes:

(1) The Importer,
(2) The Exporter,
(3) The Breeder, and
(4) The Monitor.

Figure 2 shows the four processes at each of the 64 processing nodes (as ovals).

3.3 Communication between Boss and Monitor

Communication between the Boss process of the Boss node and the Monitor process of the individual processing nodes is of the following types:

(1) The Boss process of the Boss node can initiate a Start-Up message to the Monitor process of the individual node.

(2) The Monitor process of a processing node can initiate the following four types of messages to the Boss process of the Boss node by passing along the following four types of information that it receives from the Breeder process of that node:

(a) *End-of-Generation*: The end-of-generation message contains the best-of-generation individual for the current subpopulation on the processing node and statistics about that individual such as its fitness and number of hits. This message also contains the fitness (and hits) for the worst-of-generation individual of the processing node, the average fitness (and hits) for the subpopulation as a whole, and the variance of the fitness (and hits) of the subpopulation as a whole.

(b) *Eureka*: The eureka message announces that the processing node has just created an individual in its subpopulation that satisfies the success criterion of the problem and contains the just-created best-of-run individual and statistics about it.

(c) *Trace*: The trace message announces that the Breeder process has reached certain milestones in its code (e.g., received its start-up message, completed creation of the initial random subpopulation for the node).

(d) *Error*: The error message announces that the Breeder process has encountered certain anticipatable error conditions.

3.4 The Monitor Process

The Monitor process of each processing node is continually awaiting messages from both the Boss process of the Boss node as well as from the Breeder process of its processing node. Upon receipt of a Start-Up message from the Boss, the Monitor process passes this message along to the Breeder process on its node. The Monitor process also passes messages from the Breeder process of its node along to the Boss process on the Boss node.

3.5 The Breeder Process

Several dozen parameters for controlling the run are passed from the Monitor process to the Breeder process of the node as part of the Start-Up message, including the following:

(1) the size of the subpopulation that is to be created at the processing node,

(2) the control parameters for creating the initial random subpopulation at that processing node, including the method of generation and the maximum size for the initial random individuals (and each function-defining and result-producing branch thereof),

(3) the common, network-wide table of random constants for the problem (if any),
(4) the control parameters specifying the number of each genetic operation (e.g., reproduction, crossover, etc.) to be performed for each generation,
(5) the perpetual run number for the run,
(6) a node-specific seed for the randomizer (which is used in conjunction with the perpetual run number to seed the randomizer for creating the initial random population at the node),
(7) one of the following:
  (a) the actual fitness cases for the problem, if the fitness cases are being supplied from an outside database (e.g., protein sequences or economic data),
  (b) the common network-wide randomizer seed for creating fitness cases for the run, if the fitness cases for the problem are being randomly created at the processing node,
  (c) nothing, if the fitness cases for the problem are being created programatically at the processing node (e.g., all 2^k fitness cases for a problem of symbolic regression of a k-argument Boolean function).
(8) the targeted maximum number of generations to be run and the absolute maximum number of generations to be run at the processing node (e.g., 120% of the former), and
(9) the number of primed individuals (and, if there are any, the primed individuals themselves).

After the Breeder process of a processing node receives the start-up message, it performs the following steps in connection with generation 0:

(1) Creates the initial random subpopulation of individuals for the node.
(2) Create the fitness cases for the problem at the processing node (unless the fitness cases are transmitted from the Boss process).

In the main generational loop, the Breeder process of a processing node iteratively performs the following steps:

(1) Evaluates the fitness of every individual in the subpopulation at the node.
(2) Selects, probabilistically, based on fitness, a small number of individuals to be emigrants (except on generation 0) and sends them to a buffer in the Exporter process of the node.
(3) Assimilates the immigrants currently waiting in the buffers of the Importer process of the processing node (except on generation 0). Note that the fitness of an individual is part of the data structure associated with that individual so that the fitness of an immigrant need not be recomputed upon its arrival at its destination.
(5) Creates an end-of-generation report for the subpopulation at this node for this generation.
(6) Perform the genetic operations on the subpopulation at the processing node.

The amount of computer time required to evaluate individuals in genetic programming usually varies considerably among small subpopulations. The presence of just one or a few time-consuming programs in a particular subpopulation can dramatically affect the amount of computer time required to run one generation on that node. Any attempt to synchronize the activities of the algorithm at the various processing nodes would require slowing every processing node to the speed of the slowest node. Therefore, each processing node operates asynchronously from all other processing nodes; and, after a few generations, the various processing nodes of the system will typically be simultaneously working on different generations.

This variation arises from numerous factors, including the different sizes of the individual programs, the mix of faster and slower primitive functions in the programs, and, most importantly, the number, nature, and content of the function-defining branches of the overall program. Each invocation in a calling branch of an automatically defined function requires execution of the body of that automatically defined function, so that the effective size of the overall program at the time of execution is considerably larger than the visible number of points (i.e., functions and terminals) actually appearing in the overall program. Moreover, when one automatically defined function can hierarchically refer to another, the effective size (and the execution time) of the program may be an exponential function of the visible number of points actually appearing in the overall program.

In addition, if the programs in the population are architecturally diverse (Koza 1994a), additional variation is introduced. This variation is further magnified if the architecture of a program changes during the run as a result of architecture-altering operations (Koza 1994b). For problems involving a simulation of behavior over many time steps, many separate experiments, or many probabilistic scenarios, some programs may finish the simulation considerably earlier or later than others. Indeed, for some problems (e.g., time-optimal control problems), variation in program duration is the very objective of the problem.

The entire network runs until one individual is created that satisfies the success predicate of the problem or until every processing node has completed an originally targeted maximum number of generations to be run. A processing node that reaches the originally targeted maximum number of generations before all the other nodes have reached that originally targeted generation is permitted to continue running up to an absolute maximum number of generations (usually 120% of the originally targeted maximum number).

It is inadvisable in a parallel system to attempt to check globally for the accidental creation of duplicates at the initial random generation.

3.6 The Exporter Process

The Exporter process periodically interacts with the Breeder process of its processing node as part of the Breeder's main generational loop for each generation (except generation 0). At that time, the Breeder sends four boatloads of emigrants to a buffer of the Exporter process.

The Exporter process then sends one boatload of emigrants to the Importer process of each of the four neighboring processing nodes of the network.

3.7 The Importer Process

When a boatload of immigrants arrives via any one of the four channels connecting the Importer process to its four neighboring Exporter processes, the Importer consumes the immigrants from that channel and places the immigrants into the buffer associated with that channel.

When the Breeder process of a particular processing node is ready to assimilate immigrants, zero, one, two, three, or four of the four buffers associated with its Importer process may contain new immigrants. If all four buffers are full, the four boatloads of immigrants replace the emigrants that were just dispatched by the Breeder process to the Exporter process of the node. If less than four buffers are full, the new immigrants replace as many of the just-dispatched emigrants as possible. The result is that a small number of the just-dispatched emigrants may become duplicated and exist in both this processing node and a neighboring node.

It is also possible that one of the neighbors of a particular processing node may complete processing of two generations while the given processing node completes only one generation. That neighbor will therefore have sent two boatloads of immigrants to the given node. Inasmuch as these later emigrants are the product of additional evolution at the sending node, the latest immigrants from a particular neighbor overwrite the previous immigrants from that neighbor.

3.8 The Visual Basic Monitoring Process on Host Computer

A monitoring process (written in Visual Basic) on the Host computer displays information about the run on the video display monitor of the Host computer.

One window of this display consists of a multi-color grid displaying information about the last reporting generation for each of the 64 processing nodes. The number of hits for the best-of-generation individual is shown for each node; the nodes with the best and worst number of hits are highlighted. Each processing node is colored green, yellow, red, and gray depending on the amount of time since the last end-of-generation report from that node (based on problem-specific timing parameters). Green indicates that a report was received reasonably recently based on our expectations; yellow indicates that the processing node is slower than our expectations; red indicates that the processing node is very tardy; and black indicates that the node is dead. The fastest and slowest nodes are highlighted.

A second window shows, by generation, the best fitness and best number of hits for the system as a whole. A third window shows a histogram of the number of hits.

4. Comparison of Computational Effort for Different Migration Rates

The problem of symbolic regression of the Boolean even-5-parity function will be used to illustrate the operation of the parallel system and for purposes of comparing the computational effort associated with different migration rates between the processing nodes.

The Boolean even-5-parity function takes five Boolean arguments and returns `T` if an even number of its arguments are `T`, but otherwise returns `NIL`. The terminal set, $\mathcal{T}_{adf}$, for this problem is {`D0`, `D1`, `D2`, `D3`, `D4`}. The function set, $\mathcal{F}_{adf}$, is {`AND`, `OR`, `NAND`, `NOR`} when automatically defined functions are not used. The standardized fitness of a program measures the sum, over the 32 fitness cases consisting of all combinations of the five Boolean inputs, of the Hamming-distance errors between the program and the correct Boolean value of the even-5-parity problem.

Numerous runs of this problem with a total population size of 32,000 were made using five different approaches. This particular population size (which is much smaller than the population size that can be supported on the parallel system) was chosen because it was possible for us to run this size of population on a single serial computer (a Pentium) with panmictic selection (i.e., where the individual to be selected to participate in a genetic operation can be from anywhere in the population) and thereby compare serial versus parallel processing using a common population size of 32,000. The five approaches are as follows:

(1) runs on a single serial processor with a single panmictic population of size $M = 32{,}000$,

(2) runs on the parallel system with $D = 64$ demes, a population size of $Q = 500$ per deme, and a migration rate (boatload size) of $B = 12\%$ (in each of four directions on each generation of each deme),

(3) runs as in (2) with $B = 8\%$,

(4) runs as in (2) with $B = 4\%$, and

(5) runs as in (2) with $B = 1\%$.

The maximum targeted number of generations, G, was set to 76. The size of the individual programs at the time of creation of the initial random population was limited to 500 points. Other minor parameters for the runs were selected as in Koza 1994a.

A run of this problem is considered successful when it creates a program that achieves a standardized fitness of zero (i.e., finds a program that mimics the even-5-parity function for every combination of inputs).

A probabilistic algorithm may or may not produce a successful result on a given run. Since some runs may have infinite (or unmeasurably large) duration, it is not possible to use a simple arithmetic average to compute the expected number of fitness evaluations required to solve a given problem. One way to measure the performance of a probabilistic algorithm is to use performance curves (as described in Koza 1994a). Performance curves provide a measure of the computational effort required to solve the problem with a certain specified probability (say, 99%).

The methodology for creating performance curves for runs on a parallel computing system in which the processing nodes operate asychronously differs somewhat from the methodology for runs with panmictic selection.

For the panmictic population on a single serial processor, the number of fitness evaluations, x, that are performed on a particular run is simply $x = M(i+1)$, where M is the population size and i is the generation number on which the solution emerged. However, in a parallel computing system in which the processing nodes operate asychronously, the number of fitness evaluations is

$$x = Q \, \Sigma \, (i(d) + 1),$$

where the summation index d runs over the D processing nodes, where $Q = 500$ is the subpopulation (deme) size, and where $i(d)$ is the number of the last reporting generation from processor d at the time when a program that satisfied the success criterion of the problem emerged at one processor.

Figure 3 presents the performance curves based on the 27 runs of the problem of symbolic regression of the Boolean even-5-parity function with $D = 64$ subpopulations of size $Q = 500$ and a migration rate of $B = 8\%$ (in each of four directions on each generation of each subpopulation).

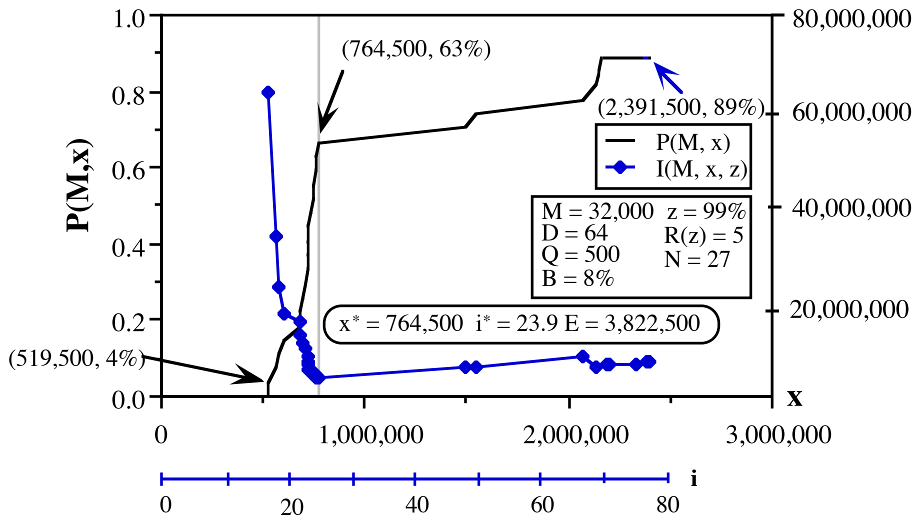

Figure 3 Performance curves for the problem of symbolic regression of the Boolean even-5-parity function with *D* = 64 demes of size *Q* = 500 and a migration rate of *B* = 8%.

The main horizontal axis of the figure represents *x*, the number of fitness evaluations performed at the time of first emergence at one processor of a program that satisfied the success criterion of the problem. The second horizontal axis (shown below the main axis) represents the equivalent number of generations, *i*, if all the processing nodes were synchronized.

The rising curve in the figure shows the experimentally observed cumulative probability of success, $P(M,x)$, of solving the problem after making *x* fitness evaluations. The left vertical axis applies to $P(M,x)$. As can be seen, the number of fitness evaluations associated with 14 of the 27 runs lies in the relatively narrow domain between 675,000 and 773,000. The first value of *x* for which $P(M,x)$ is non-zero occurs when $x = 519{,}500$ fitness evaluations (equivalent to 16.2 synchronous generations); $P(M,x)$ is 4% for that *x*. The experimentally observed value of $P(M,x)$ over the 27 runs is 63% after making 764,500 fitness evaluations and 89% after making 2,391,500 fitness evaluations.

The second curve in the figure shows the number of individuals that must be processed, $I(M,x,z)$, to yield, with probability *z*, a solution to the problem after making *x* fitness evaluations. The right vertical axis applies to $I(M,x,z)$. $I(M,x,z)$ is derived from the experimentally observed values of $P(M,x)$ as the product of the number of fitness evaluations, *x*, and the number of independent runs, *R(z)*, necessary to yield a solution to the problem with a required probability, *z*, after making *x* fitness evaluations. In turn, *R(z)* is given by

$$R(z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,x))} \right\rceil,$$

where the brackets indicate the ceiling function for rounding up to the next highest integer. The required probability, *z*, will be 99% herein.

The $I(M,x,z)$ curve reaches a minimum value of 3,822,500 at the best value $x^* = 764{,}500$ (equivalent to $i^* = 23.9$ synchronous generations) as shown by the shaded vertical line. For the observed value of $P(M,x^*) = 63\%$ associated with the best x^*, the number of independent runs necessary to yield a solution to the problem with a 99% probability is $R(z) = 5$. The three summary numbers ($x^* = 764{,}500$, $i^* = 23.9$, and $E = 3{,}822{,}500$) in the oval indicate making multiple runs of $x^* = 764{,}500$ fitness evaluations (equivalent to $i^* = 23.9$ synchronous

generations) is sufficient to yield a solution to this problem after making a total of E = 3,822,500 fitness evaluations with 99% probability. E is a measure of the computational effort necessary to yield a solution to this problem with 99% probability.

Table 1 shows the computation effort for each of the six approaches to solving this problem.

Table 1 Comparison of six approaches.

	Approach	Migration rate B	Computational effort E	Best x^*	$P(M,x)$ for best x^*
1	Panmictic	NA	6,912,000	1,888,000	83%
2	Parallel	12%	7,072,500	2,357,500	83%
3	Parallel	8%	3,822,500	764,500	63%
4	Parallel	4%	8,174,000	2,435,999	71%
5	Parallel	1%	6,594,000	2,198,000	89%

As can be seen, the computational effort, E, is smallest for a migration rate of B = 8%.

The computational effort for the runs with the 8% migration rate is less than the computational effort required with the panmictic population. In other words, creating semi-isolated demes produced the bonus of also improving the performance of genetic programming for this migration rate for this problem. That is, apart from the nearly linear speed-up in executing a fixed amount of code inherent in the island model of genetic programming, parallelization delivered *more than linear* speed-up in solving the problem using genetic programming. This result is consistent with the analysis of Sewell Wright (1943) regarding demes of biological populations.

For comparison, we also made 497 runs of this problem using a population size of only 500. Only 0.9% of the runs were successful. The computational effort, E, for the population size of 500 was 16,898,000, thereby demonstrating that this problem cannot be efficiently solved with the population of one node alone (i.e., with no migration).

5. Conclusions

Parallel genetic programming was successfully implemented in the C programming language on a host PC 486 computer (running Windows) and a network of 64 transputers.

Genetic programming required the least computational effort to solve the problem of symbolic regression of the Boolean even-5-parity function with an 8% migration rate. This computational effort was less than that required for a panmictic population of the same size. That is, apart from the nearly linear speed-up in executing a fixed amount of code inherent in the island model of genetic programming, parallelization delivered *more than linear* speed-up in solving the problem using genetic programming.

6. Acknowledgements

James P. Rice helped develop the design criteria and design for the parallel computer system described herein. Walter Alden Tackett did the early design work on the various programming processes. Hugh Thomas of SGS-Thompson Microelectronics wrote configuration scripts and assisted in debugging the program and systems. Simon Handley made numerous helpful comments on this report.

7. Bibliography

Abramson, David, Mills, Graham, and Perkins, Sonya. 1994. Parallelisation of a genetic algorithm for the computation of efficient train schedules. In Arnold, David, Christie, Ruth, Day, John, and Roe, Paul (editors). *Parallel Computing and Transputers*. Amsterdam: IOS Press. Pages 139–149.

Bianchini, Ricardo and Brown, Christopher. 1993. Parallel genetic algorithms on distributed-memory architectures. In Atkins, S. and Wagner, A. S. (editors). *Transputer Research and Applications 6*. Amsterdam: IOS Press. Pages 67–82.

Cui, J. and Fogarty, T. C. 1992. Optimization by using a parallel genetic algorithm on a transputer computing surface. In Valero, M, Onate, E., Jane, M., Larriba, J. L., Suarez, B. (editors). *Parallel Computing and Transputer Applications*. Amsterdam: IOS Press. Pages 246–254.

Dietz, H. G. 1992. Common subexpression induction. Parallel Processing Laboratory Technical Report TR-EE-92-5. School of Electrical Engineering. Purdue University.

Dietz, H. G. and Cohen, W. E. 1992. A Massively parallel MIND implemented by SIMD hardware. Parallel Processing Laboratory Technical Report TR-EE-92-4. School of Electrical Engineering. Purdue University.

Goldberg, David E. l989a. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Goldberg, David E. 1989b. Sizing populations for serial and parallel genetic algorithms. In Schaffer, J. D. (editor). *Proceedings of the Third International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 70-79.

Holland, John. H. 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press. Also Cambridge, MA: The MIT Press 1992.

Juric, M., Potter, W. D., and Plaksin, M. 1995. Using the parallel virtual machine for hunting snake-in-the-box codes. In Arabnia, Hamid R. (editor) *Transputer Research and Applications 7*. Amsterdam: IOS Press.

Kinnear, Kenneth. E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Koza, John. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.

Koza, John. R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press.

Koza, John R. 1994b. Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming. Stanford University Computer Science Department technical report STAN-CS-TR-94-1528. October 21, 1994.

Kroger, Berthold, Schwenderling, Peter, and Vornberger, Oliver. 1992. Massive parallel genetic packing. In Reijns, G. L. and Luo, J. (editors). *Transputing in Numerical and Neural Network Applications*. Amsterdam: IOS Press. Pages 214-230.

Min, Shermann L. 1994. Feasibility of evolving self-learned pattern recognition applied toward the solution of a constrained system using genetic programming. In Koza, John R. (editor). *Genetic Algorithms at Stanford 1994*. Stanford, CA: Stanford University Bookstore. ISBN 0-18-187263-3.

Robertson, George. l987. Parallel implementation of genetic algorithms in a classifier system. In Davis, L. (editor). *Genetic Algorithms and Simulated Annealing*. London: Pittman.

Schwehm, M. Implementation of genetic algorithms on various interconneciton networks. In Valero, M, Onate, E., Jane, M., Larriba, J. L., Suarez, B. (editors). 1992. *Parallel Computing and Transputer Applications*. Amsterdam: IOS Press. Pages 195-203.

Singleton, Andrew. 1994. Personal communication.

Stender, Joachim (editor). 1993. *Parallel Genetic Algorithms*. Amsterdam: IOS Publishing.

Tanese, Reiko. 1989. *Distributed Genetic Algorithm for Function Optimization*. PhD. dissertation. Department of Electrical Engineering and Computer Science. University of Michigan.

Tout, K. Ribeiro-Filho, B, Mignot, B, and Idlebi, N. A. 1994. cross-platform parallel genetic algorithms programming environment. *Transputer Applications and Systems '94*. Amsterdam: IOS Press. 1994. Pages 79–90.

Wright, Sewall. 1943. *Genetics* 28. Page 114.

Computer Science Department January 30, 1995

PARALLEL GENETIC PROGRAMMING ON A NETWORK OF TRANSPUTERS

John R. Koza

David Andre

Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, California 94305–2140 USA

PARALLEL GENETIC PROGRAMMING ON A NETWORK OF TRANSPUTERS

John R. Koza
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, California 94305-2140 USA
Koza@CS.Stanford.Edu
Phone: 415-941-0336
FAX: 415-941-9430

David Andre
Visiting Scholar
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, California 94305-2140 USA
andre@flamingo.stanford.edu
Phone: 415-326-5113

ABSTRACT

This report describes the parallel implementation of genetic programming in the C programming language using a PC 486 type computer (running Windows) acting as a host and a network of transputers acting as processing nodes. Using this approach, researchers of genetic algorithms and genetic programming can acquire computing power that is intermediate between the power of currently available workstations and that of supercomputers at a cost that is intermediate between the two.

A comparison is made of the computational effort required to solve the problem of symbolic regression of the Boolean even-5-parity function with different migration rates. Genetic programming required the least computational effort with an 8% migration rate. Moreover, this computational effort was less than that required for solving the problem with a serial computer and a panmictic population of the same size. That is, apart from the nearly linear speed-up in executing a fixed amount of code inherent in the parallel implementation of genetic programming, parallelization delivered *more than linear* speed-up in solving the problem using genetic programming.