David Andre and John R. Koza

This chapter describes the parallel implementation of genetic programming in the C programming language using a PC type computer (running Windows) acting as a host and a network of processing nodes using the transputer architecture. 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. This approach is illustrated by a comparison 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 5% 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, the use of distributed sub-populations with only limited migration delivered *more than linear* speed-up in solving the problem.

16.1 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 populations. Since the computational burden of the genetic algorithm is proportional to the population size, 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 many parallel supercomputers are, of course, expensive and may not be accessible. Therefore, we focus our attention on other more flexible, general-purpose, and inexpensive methods of parallelization.

Section 16.2 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 16.3 then describes the successful parallel implementation of genetic programming in the C language using a PC 486 type computer and a network of transputers. Section 16.4 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 16.5 describes the successful implementation of a variant of our parallel system where a PowerPC is joined with a transputer at each node of the network, maintaining the transputer communication architecture but vastly increasing the computational power of the system.

16.2 Selection of Machinery to Implement Parallel Genetic Programming

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

16.2.1 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 most practical problems, the dominant consideration as to computer time for the genetic algorithm is the evaluation of the fitness 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 fitness 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 usually 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 usually constitute a major burden 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).

16.2.2 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., 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 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 inter-processor 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 of many individuals.

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.

16.2.3 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 above 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 up to 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 may approach three 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, and 100 such generations would occupy about a day.

Our design criteria were thus 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. "Price" thus also reflected our desire to minimize the amount of time spent on the task of parallelization itself, and maximize the time spent on genetic programming research.

16.2.4 Qualitative Evaluation of Several Parallel Systems

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, an 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 as far as the simplicity of implementation, 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.

The fine-grained SIMD ("Single Instruction, Multiple Data") supercomputers (such as Thinking Machine's CM-2 and MasPar's machines) are largely inappropriate for genetic programming because the individual programs in the population are generally of different sizes and shapes and contain numerous conditional branches. Although 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], doing so would require substantial expertise outside our existing capabilities, as well as extensive customizing of the parallel code for each new problem.

The traditional 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 capability. 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.

Given the high expense and inappropriate features of many of the existing parallel supercomputers and serial supercomputers, we looked into the possibility of simpler, more general-purpose parallel systems built out of a network of simple processor boards or workstations.

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.

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 some difficulties involving the ongoing management and maintenance of the independent processes on a heterogeneous collection of machines. Although it is possible to use independent computers in this manner [Singleton 1994], these computers (and most of the supporting software) were designed for independent operation, and there are no simple tools for the design, configuration, routing, or bootloading of the many interconnected processes that must run on a parallel machine.

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.

16.2.5 Transputers

A transputer is a single VLSI device containing a 32-bit 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 and multitasking. Transputers are designed to operate in parallel and their on-chip communication links and multiprocessing capabilities were specifically designed to facilitate parallel computing. Transputers are typically mounted on small boards, called TRAMs, with up to 16 megabytes of additional RAM memory. One currently popular model of TRAM (occupying approximately the volume of a small pile of 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.

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 either within the PC computer (if it has adequate power supply and ventilation), or 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 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 nodes. Moreover, the set of tools provided by INMOS for programming the transputer is very simple to use, and hides most of the thorny issues of parallelization from the user. The toolkit allows the user to specify a virtual topology of the network i.e., what processes and channels there are, and how they are interconnected. The toolkit then places the virtual topology onto the physical hardware. In addition, the toolkit provides an advanced bootloader, which loads the executables onto each of the nodes of the network in a very general way. The toolkit's debugger is advanced and allows post-mortem analysis of any of the processes on the network.

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; Schwehm 1992; Tout, Ribeiro-Filho, Mignot, and Idlebi 1994; Juric, Potter and Plaksin 1995].

16.2.6 Suitability of a Transputers for Parallel Genetic Programming

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 each of the four directions. If 8% of the individuals at each processing node are selected as emigrants in each of four directions and the population size is 1000, each of these four boatloads of emigrants would contain 80 2,500-byte individuals. The communication capability of 20 megabits per second is obviously more than sufficient to handle 200,000 bytes every 15 minutes (the expected time for one generation).

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. Although a single 486/33 is obviously currently not the fastest microprocessor, 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 (or about 3.0 megabytes, if the population happens to be architecturally diverse, see Section 16.2.3). Thus, on a 4-megabyte TRAM, approximately 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.

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. 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.

16.3 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.

16.3.1 The Physical System

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

• the host computer consisting of a keyboard, a video display monitor, and a large disk memory,

• a transputer running the debugger,

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

• the 64 transputers of the network, each running a Monitor process, a Breeder process, an Exporter process, and an 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 a VME box. 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 other processing nodes. 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 the physical links using a virtual router provided in the INMOS Toolkit. Figure 16.1 shows the various elements of the system.

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.

16.3.2 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 boss. The second process on the Host computer is the monitoring process.

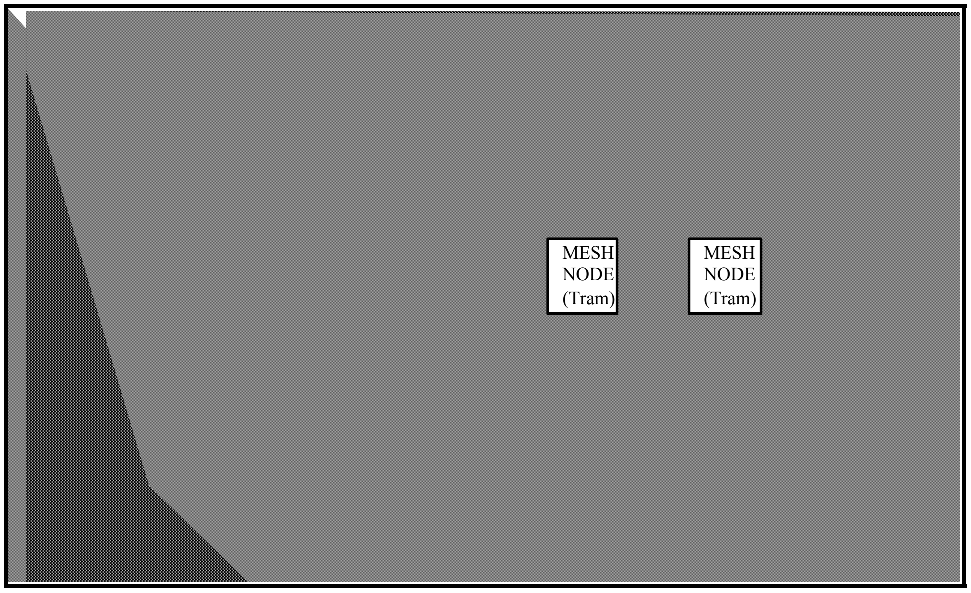

Figure 16.1
The boxes in the figure denote the various computers, including the host computer (a PC 486 type machine), the Debugger Node (a TRAM), the Boss Node (a TRAM), and the network of processing nodes (TRAMs), which are also called mesh-nodes. For simplicity, the figure shows only a netrowk of nine nodes. The ovals denote the various files (all on the host computer), including 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, called the VB process). The rounded boxes denote the input-output devices of the host computer, including its keyboard and video display monitor. The diamond denotes the 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 TRAM with the Debugger runs only one process, the INMOS-supplied Debugger process. The TRAM with the Boss runs only one process, the Boss. Each of the 64 TRAMs in the toroidal network concurrently runs the following four processes: the Importer, the Exporter, the Breeder, and the Monitor (See Figure 16.2). The primary process on each of the processing nodes is the Breeder process. The other three processes permit asynchronous communication and serve to avoid deadlocks.

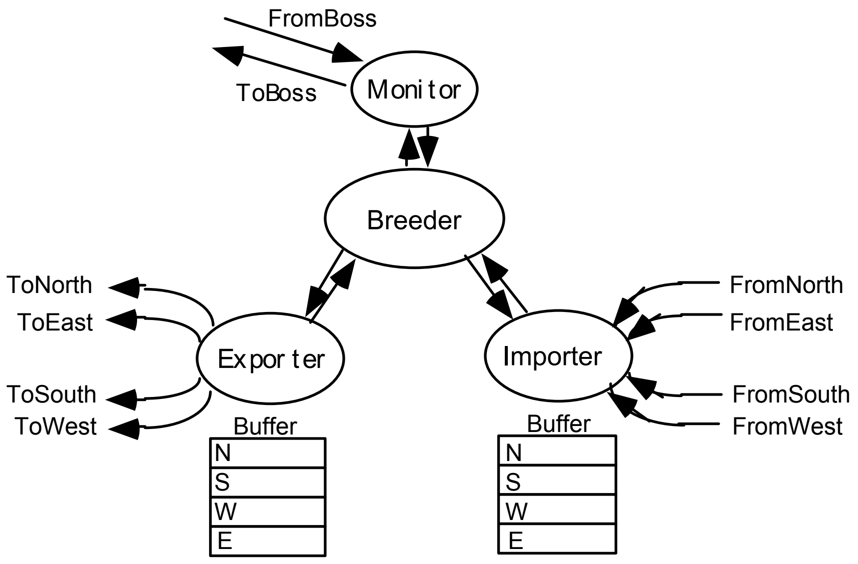

Figure 16.2
The four processes on each of the 64 processing nodes. The Breeder process performs the bulk of the computation in genetic programming, and the three other processes facilitate communication with the Boss and the four neighboring nodes.

16.3.3 The Boss Process

The Boss process is responsible for communicating with the Host process, for sending initial start messages to each of the processors in the network, for tabulating information sent up from each processor at the end of each generation, for sending information to the monitoring process on the host, and for handling error conditions.

At the beginning of a run of genetic programming, the Boss process initializes various data structures, creates the set of fitness cases either functionally or by obtaining information from a file on the Host, creates a different random seed for each processor, pings each processor to be sure it is ready, and reads in a set of parameters from a file on the host that controls genetic programming. Then, the Boss sends a Start-Up message to each Monitor process (which will in turn send it along to the Breeder process). This message contains the following:

• the size of the subpopulation that is to be created at the processing node,

• 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),

• the common, network-wide table of random constants for the problem (if any),

• the control parameters specifying the number of each genetic operation (e.g., reproduction, crossover, etc.) to be performed for each generation,

• a node-specific seed for the randomizer,

• the actual fitness cases for the problem,

• the number of primed individuals (and, if any, the primed individuals themselves).

After sending the Start-up message, the Boss enters a loop where it handles the various messages that each Monitor sends until an error condition occurs, a solution is found, or all processors have either crashed or completed a number of generations specified in a parameter file.

16.3.4 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 the following messages from the Breeder process of its node along to the Boss:

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 at the node, and the variance of the fitness (and hits) of the subpopulation.

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 various statistics about it.

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).

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

16.3.5 The Breeder Process

After the Breeder process of a processing node receives the Start-up message, it creates the initial random subpopulation of individuals for the node. Then, 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.
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.
3. Assimilates the immigrants currently waiting in the buffers of the Importer process (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.
4. Creates an end-of-generation report for the subpopulation.
5. Performs the genetic operations on the subpopulation.

The breeder process runs until one individual is created that satisfies the success predicate of the problem or until it is stopped by the Boss process.

16.3.6 Asynchronous operation

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. 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. Therefore, each processing node operates asynchronously from all other processing nodes. After a few generations, the various processing nodes of the system will typically be 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 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, 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 desired asynchrony of the generations on nearby processors requires that the exporting and importing of migrating programs take place in a manner that does not require that the breeder itself ever wait for a neighboring process to finish a generation. To allow the breeder nearly uninterrupted computing time, the Exporter process and the Importer process were created to handle the communication. The Monitor process acts in a similar fashion for communication with the Boss process. In addition, the use of multiple processes is important to prevent dead-locks from taking place.

16.3.7 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.

16.3.8 The Importer Process

The purpose of the Importer is to store incoming boatloads of emigrants until the Breeder is ready to incorporate them into the subpopulation. 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 these immigrants into the buffer associated with that channel (occasionally overwriting previously arrived, but not yet assimilated, immigrants in that buffer). When the Breeder process is ready to assimilate immigrants, it calls for the contents of the Importer's buffers. 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 fewer than four buffers are full, the new immigrants replace as many of the just-dispatched emigrants as possible.

16.4 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 to compare 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 result produced by the program and the correct Boolean value of the even-5-parity function. For a full description of, and tableau for, this problem, see Koza [1994a].

Numerous runs of this problem with a total population size of 32,000 were made using ten 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 ten approaches are as follows:

1. runs on a single serial processor with a 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. - 10. runs as in (2) with $B = 8\%$, 6%, 5%, 4%, 3%, 2%, 1%, and 0%, respectively.

The number of generations, G, that each breeder process must complete unless a solution is found, 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., the program mimics the even-5-parity function for all 32 combinations 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 asynchronously 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 asynchronously, the number of fitness evaluations is

$$Q \sum_{d=1}^{D} [i(d) + 1] \tag{16.1}$$

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 16.3 presents the performance curves based on the 34 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 $B = 5\%$ (in each of four directions on each generation of each subpopulation).

The main horizontal axis of the figure represents x, the number of fitness evaluations performed at the time of the 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 $P(M,x)$. The first value of x for which $P(M,x)$ is non-zero occurs when $x = 1{,}202{,}000$ fitness evaluations; $P(M,x)$ is 3% for that x. The experimentally observed value of $P(M,x)$ over the 34 runs is 94% after making 1,671,000 fitness evaluations and 97% after making 2,417,500 fitness evaluations.

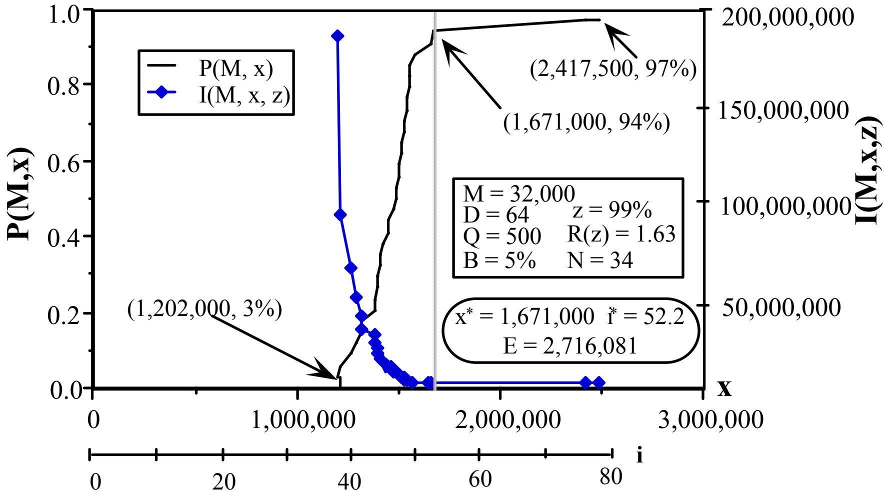

Figure 16.3
Performance curves for the problem of symbolic regression of the Boolean even-5-parity function with *D* = 64 demes of size *Q* = 500 and a migration rate of *B* = 5%. Details of the figure are explained in section 16.4 of the text.

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

$$R(z) = \frac{\log(1 - z)}{\log(1 - P(M,x))} \tag{16.2}$$

The required probability, *z*, will be 99% herein. This equation for *R(z)* differs from the equation used in previous work [Koza 1992, 1994a], in that the value of *R(z)* is not rounded to the nearest larger integer, but rather is kept as a fractional value.

The *I*(*M*,*x*,*z*) curve reaches a minimum value of 2,716,081 at the best value *x** = 1,671,000 (equivalent to *i** = 52.2 synchronous generations) as shown by the shaded vertical line. For the observed value of $P(M,x^*)$ = 94% associated with the best x^*, the number of independent runs necessary to yield a solution to the problem with a 99% probability is $R(z) = 1.63$. The three summary numbers (x^* = 1,671,000, i^* = 52.2, and E = 2,716,081) in the oval indicate making multiple runs of x^* = 1,671,000 fitness evaluations (equivalent to i^* = 52.2 synchronous generations) is sufficient to yield a solution to this problem after making a total of E = 2,716,081 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 the ten approaches to solving this problem.

As can be seen, the computational effort, E, is smallest for a migration rate of B = 5%. The computational effort for all of the parallel runs except for those runs with an extreme (0% or 12%) 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 these migration rates for this problem. The parallel system has two speed-ups: the nearly linear speed-up in executing a fixed amount of code inherent in the island model of genetic programming and the *more than linear* speed-up in terms of the speed of the genetic programming algorithm in solving the problem. In other words, not only does the problem solve more quickly because we are distributing the computing effort over many processors, but there is less computational effort needed because of the multiple populations. This result is consistent with the analysis of Sewell Wright [1943] regarding demes of biological populations. Subpopulations can, of course, also be created and used on a serial

Table 16.1
Comparison of ten approaches.

	Approach	Migration rate B	Computational effort $I(M,x,z)$
1	Panmictic	NA	5,929,442
2	Parallel	12%	7,072,500
3	Parallel	8%	3,822,500
4	Parallel	6%	3,078,551
5	**Parallel**	**5%**	**2,716,081**
6	Parallel	4%	3,667,221
7	Parallel	3%	3,101,285
8	Parallel	2%	3,822,500
9	Parallel	1%	3,426,878
10	Parallel	0%	16,898,000

machine.

16.5 Increasing Performance: The PowerPC in the Transputer Architecture

One of the most important advantages of the transputer architecture is the ease of implementing an algorithm in a parallel environment. However, the transputer, although a moderately fast processor, is not particularly powerful by today's standards. Several companies have capitalized on the success of the transputer architecture for communication and multiprocessing in a parallel environment by creating hybrid systems that combine a more powerful microprocessor (such as the DEC Alpha or the Motorola PowerPC) with the transputer architecture and the INMOS toolkit. These companies use the transputer microprocessor solely for communication.

One such company, Parsytec of West Germany, has produced several commercial parallel processing systems that integrate the PowerPC chip with the transputer architecture. The basis of their systems are TRAM-like components that combine a PowerPC chip with a transputer, some memory, and some communication hardware. Their system utilizes the INMOS toolkit in its entirety, with several special add-on libraries and tools to handle the additional processor. In essence, their system delivers a increase in performance over the transputer-only system, but requires little change in the architecture, because the INMOS toolkit eliminates most of the vexatious details.

Our second implementation of parallel genetic programming has almost the same architecture as our first implementation. It consists of a network of 64 PowerPC TRAMs (containing both a PowerPC 601 processor, a transputer T805 processor, and 32 megabyte of RAM), one transputer-only TRAM as the Boss, and a PC 586/90 type computer acting as host. Each of the 64 PowerPC and transputer nodes act as the processing nodes of the network. The single transputer T805 TRAM with 16 MB of RAM acts as the Boss node. The debugger node has been eliminated.

A good example of the ease of use of INMOS' parallel toolkit is the fact that the entire port of our parallel implementation to the new Parsytec system took less than a week, including the system setup, learning several tools to analyze the PowerPC nodes in the network, and the porting, compiling, and debugging of all the code in the parallel genetic programming kernel and our benchmark problems. In less than a week, we were able to run the 5-parity problem on our PowerPC network and compare it to the performance of the transputer system.

On several runs of the 5-parity problem, we found that the Parsytec PowerPC and transputer system outperformed the transputer system by a factor of more than 22. However, the cost of each node in the PowerPC and transputer system cost only approximately five times more than a node in the first implementation. Our second implementation thus provides a significantly better price-to-performance ratio over our first system. In addition to greater processing speed, our second implementation gives us the capability to evolve populations that are two orders of magnitude larger than the populations evolved in most previous work on genetic programming [Koza 1992, 1994a; Kinnear 1994]. On the 6-parity problem, for example, we were able to evolve populations of two million 750-point individuals.

16.6 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. In addition, the simplicity of the transputer architecture allowed the successful implementation of a second system that combined the computational power of the PowerPC with the parallel processing abilities of the transputer. Both of our implementations indicate that the transputer architecture allows a powerful, easily expandable, parallel system for genetic programming that is intermediate in cost between stand-alone workstations and expensive supercomputers.

Genetic programming required the least computational effort to solve the problem of symbolic regression of the Boolean even-5-parity function with an 5% 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.

Acknowledgments

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 chapter.

Bibliography

Abramson, D., Mills, G., and Perkins, S. (1994). Parallelization of a genetic algorithm for the computation of efficient train schedules. In Arnold, D., Christie, R., Day, J., and Roe, P. (editors). *Parallel Computing and Transputers*. Amsterdam: IOS Press. p 139–149.

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 MIMD implemented by SIMD hardware. Parallel Processing Laboratory Technical Report TR-EE-92-4. School of Electrical Engineering. Purdue University.

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

Goldberg, D. 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. p 70-79.

Holland, J. 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, H. R. (editor) *Transputer Research and Applications 7*. Amsterdam: IOS Press.

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

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

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

Koza, J. 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.

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

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

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

Singleton, A. (1994). Personal communication.

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

Tanese, R. (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). A cross-platform parallel genetic algorithms programming environment. *Transputer Applications and Systems '94*. Amsterdam: IOS Press. 1994. p 79–90.

Wright, Sewall. (1943). *Genetics* 28. p 114.

PARALLEL GENETIC PROGRAMMING: A SCALABLE IMPLEMENTATION USING THE TRANSPUTER ARCHITECTURE

David Andre

Visiting Scholar
Computer Science Department
Stanford University
860 Live Oak Ave, #4
Menlo Park, CA 94025 USA
EMAIL: andre@flamingo.stanford.edu
PHONE: 415-326-5113
WWW: http://www-leland.stanford.edu/~phred/

John R. Koza

Computer Science Department
Stanford University
Stanford, CA 94305-2140 USA
EMAIL: Koza@CS.Stanford.Edu
PHONE: 415-941-0336
FAX: 415-941-9430
WWW: http://www-cs-faculty.stanford.edu/~koza/