A PARALLEL IMPLEMENTATION OF GENETIC PROGRAMMING THAT ACHIEVES SUPER-LINEAR PERFORMANCE

David Andre
Visiting Scholar, Computer Science Department
Stanford University
EMAIL: andre@flamingo.stanford.edu
WWW: http://www-leland.stanford.edu/~phred/

John R. Koza
Computer Science Department
Stanford University
EMAIL: Koza@CS.Stanford.Edu
WWW: http://www-cs-faculty.stanford.edu/~koza/

Abstract: This paper describes the successful parallel implementation of genetic programming on a network of processing nodes using the transputer architecture. With 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 intermediate cost. This approach is illustrated by a comparison of the computational effort required to solve a benchmark problem. Because of the decoupled character of genetic programming, our approach achieved a nearly linear speed up from parallelization. In addition, for the best choice of parameters tested, the use of subpopulations delivered a *super linear* speed-up in terms of the ability of the algorithm to solve the problem. Several examples are also presented where the parallel genetic programming system evolved solutions that are competitive with human performance on the same problem.

1 Introduction

Amenability to parallelization is an appealing feature of genetic algorithms, evolutionary programming, evolution strategies, classifier systems, and genetic programming [1][2][3][4].

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 single-processor 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 are, of course, expensive and may not be accessible. Therefore, we focus our attention on other more flexible, scalable, and inexpensive methods of parallelization.

Section 2 presents a brief overview of genetic programming. Section 3 presents the factors that caused us to choose the transputer architecture. Section 4 then describes the successful parallel implementation of genetic programming. Section 5 compares the computational effort required to

solve a problem using various migration rates. Section 6 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. Section 7 briefly describes four problems upon which genetic programming, running on this parallel architecture, has evolved solutions that are at or exceed human levels of performance on the same problem.

2 Background on genetic programming and genetic algorithms

Genetic programming *is* automatic programming. The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [5] provides evidence that genetic programming can solve, or approximately solve, a variety of problems from a variety of fields, including many benchmark problems from machine learning, artificial intelligence, control, robotics, optimization, game playing, symbolic regression, and concept learning. Genetic programming is an extension of Holland's genetic algorithm described in [1].

Genetic programming starts with a primordial ooze of randomly generated computer programs composed of the available programmatic ingredients and then applies the principles of animal husbandry to breed a new (and often improved) population of programs. The breeding is done in a domain-independent way using the Darwinian principle of survival of the fittest, an analog of the naturally-occurring genetic operation of crossover (sexual recombination), and occasional mutation. The crossover operation is designed to create syntactically valid offspring programs (given closure amongst the set of ingredients). Genetic programming combines the expressive high-level symbolic representations of computer programs with the near-optimal efficiency of learning of Holland's genetic algorithm. A program that solves (or approximately solves) a given problem often emerges from this process. Many recent advances in genetic programming are described in [6], [7], [8], and [9].

3 Selection of machinery to implement parallel genetic programming

This section describes our search for a relatively simple, practical, and scalable method for implementing parallel genetic programming.

3.1 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 and do not consume large amounts of computer time. Thus, for most practical problems, the dominant factor with respect to computer time is the evaluation of the fitness of each individual in the population. Fortunately, these time-consuming fitness evaluations are totally decoupled from one another and can be performed independently for each individual in the population.

The dominant factor with respect 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 on 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).

3.2 The island model of parallelization

In a coarse-grained or medium-grained parallel computer, subpopulations are situated at the separate processing nodes. This approach is called the *distributed genetic algorithm* [2] or the *island model* for parallelization. The subpopulations are often called *demes* (after [10]).

When a run of the distributed genetic algorithm begins, each processing node locally creates its own initial random subpopulation. Each processing node then measures the fitness of all 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). The offspring resulting from the genetic operations are then measured for fitness and this iterative process continues. Because the time-consuming fitness measurement is performed independently for each individual and independently at each processing node, this approach delivers an overall increase in performance that is nearly linear with the number of independent processing nodes [2].

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 each adjacent node within the toroidal topology of processing nodes. When the immigrants arrive at their destination, a like number of individuals are selected for deletion from the destination processor. 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, which is a population where the individual to be selected to participate in a genetic operation can be from anywhere in the population). 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 infrequently -- only after completion of the entire generation. The distributed genetic algorithm is well suited to loosely coupled, low bandwidth, parallel computation.

3.3 Design considerations for parallel genetic programming

The largest of the programs evolved in *Genetic Programming* [5] and *Genetic Programming II* [6] 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 parallel implementation of genetic programming described herein was specifically designed for problems whose solutions require 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 up to 2,500 points, where each point can be stored in one byte of memory. In this representation, each point represents either a terminal or a function (with a known arity). Thus, a population of 1,000 2,500-point programs can be accommodated in 2.5 megabytes of storage with this one-byte representation. 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 effort needed to implement the expected succession of new problems that we would approach with genetic programming.

3.4 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 [11] verified) that these machines would not prove to deliver particularly good performance on genetic programming applications because of the disorderly sequence of conditionals and other operations in the program trees of the individuals in the population. The Intel i860 microprocessor was also avoided for the same reason. Although a serial solution would have been ideal as far as the simplicity of implementation, we believe that greater gains in computing speed 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 that seemingly requires a MIMD machine [12][13], 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 designed to deliver a larger ratio of inter-processor communication bandwidth relative to processor speed than is optimal for genetic programming. 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 speed.

Given the high expense and inappropriate features of many of the existing parallel supercomputers and serial supercomputers, we looked into the possibility of simpler parallel systems built out of a network of 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 the constant stream of new problems to which we apply genetic programming.

We also considered the idea of networks of workstations or networks of single-board computers (each containing an Ethernet adapter); however, these approaches present difficulties involving the ongoing management and maintenance of the independent processes on many autonomous machines. Even for a network of single-board computers, the tools and operating systems used in these systems were not as sophisticated as those in the system we finally selected. Although it is possible to use independent computers in this manner [14], 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 such a network of processors.

3.5 *Transputers*

One important reason for considering the transputer (designed by INMOS, a division of SGS-Thomson Microelectronics) was that it was designed to support parallel computation involving multiple inter-communicating processes on each processor and inter-processor communication. The transputer was engineered 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.

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 [15][16][17][18][19][20][21].

3.6 *Suitability of 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 all four directions. 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. 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. On a 4-megabyte TRAM, there is, therefore, about 1.5 megabytes of memory remaining after accommodating the population. This remaining amount of memory is sufficient for storing the program for genetic programming, the communication buffers, and the stack, 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. Finally, 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.

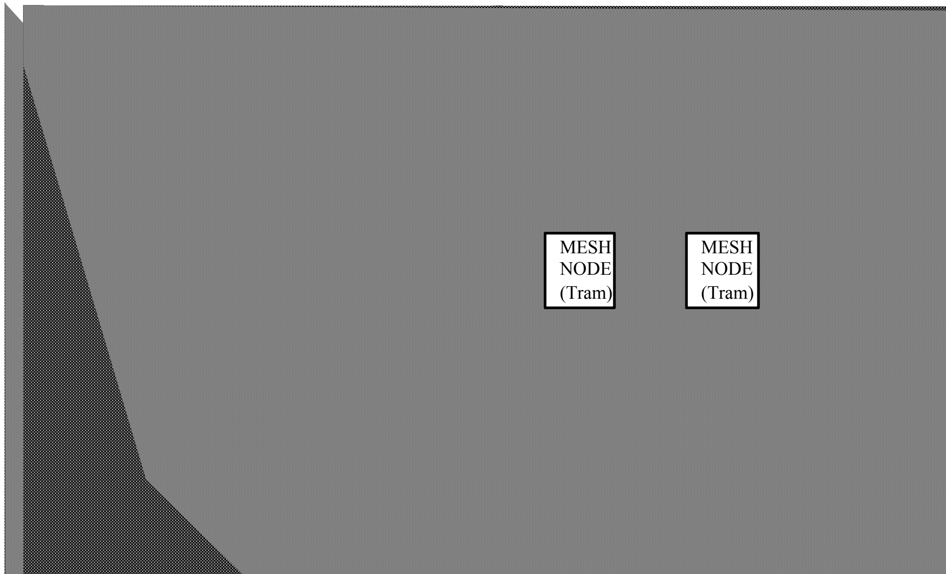

Figure 1. The boxes denote the various computers, including the host Pentium computer, the Debugger Node, the Boss Node, and the network of processing nodes. For simplicity, only nine of the 64 processing nodes are shown. The ovals denote the various major files on the host computer, including the input parameter file for each run and two output files (one containing the results of the run and one containing information needed by a monitoring process). The rounded boxes denote the input-output devices of the host computer. The diamond denotes the Visual Basic (VB) 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 virtual channels connecting each of the processing nodes with their neighbors.

4 Implementation of parallel genetic programming using a network of transputers

A network of 66 TRAMs and one Pentium 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.

4.1 The physical system

The physical system was purchased from Transtech of Ithaca, NY. A PC computer (running Windows 3.1) 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 and serve as the central supervisory process for running genetic programming (the Boss Node) and a node to run the INMOS debugger (the Debugger 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, channels. The channels are laid out along the physical links using a virtual router provided in the INMOS Toolkit. Figure 1 shows the system's various elements.

4.2 Intercommunicating 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.

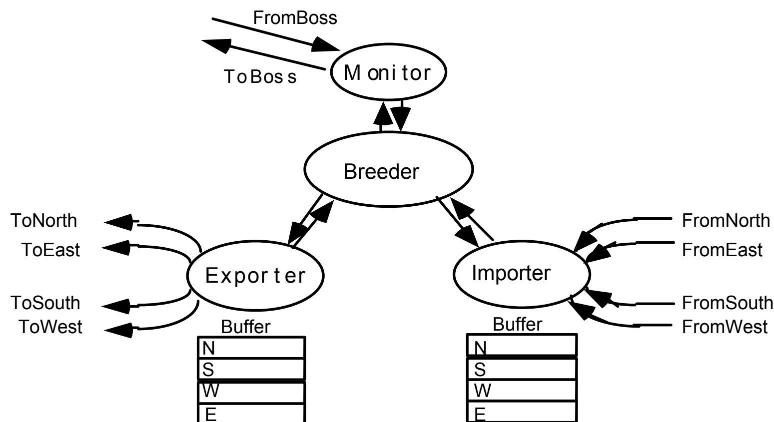

Figure 2 The four processes on each of the 64 processing nodes. The Breeder process performs the bulk of the computation in genetic programming, and the three other processes facilitate communication with the Boss and the four neighboring nodes.

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 (Figure 2). The primary process on each of the processing nodes is the Breeder process, which runs the main operations of genetic programming. The other three processes permit asynchronous communication and serve to avoid deadlocks.

4.3 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 control parameters for creating the initial random subpopulation, including the size of the subpopulation, the method of generation and the maximum size for the initial random individuals, the common, network-wide table of random constants for the problem (if any), the control parameters specifying the number of each genetic operation to be performed for each generation, and a node-specific seed for the randomizer.

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 specified number of generations.

4.4 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, statistics about that individual, and some general statistics.

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.

4.5 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).
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.

4.6 Asynchronous operation

For most problems the amount of computer time required to evaluate individuals in genetic programming usually varies considerably among 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 with respect to 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 within the programs, and the number, nature, and content of the function-defining branches of the overall program.

The 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 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. The use of multiple processes is also important in preventing dead-locks.

4.7 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.

4.8 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.

5 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, T_{adf}, for this problem is {`D0`, `D1`, `D2`, `D3`, `D4`}. The function set, F_{adf}, is {`AND`, `OR`, `NAND`, `NOR`}. 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 this problem, see [6].

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 and thereby compare serial versus parallel processing using a common population size of 32,000. The first approach tested runs on a single processor with a panmictic population of size M = 32,000; the second approach tested runs on the parallel system with D = 64 demes, a population size of Q = 500 per deme, and a migration rate of B = 12% (in each of four directions on each generation of each deme); the third through tenth approaches tested runs on the parallel system with D = 64 demes, a population size of Q = 500 per deme, and a migration rate of B = 8%, 6%, 5%, 4%, 3%, 2%, 1%, and 0%, respectively.

A probabilistic algorithm may or may not produce a successful result on a given run. One way to measure the performance of a probabilistic algorithm is to compute the computational effort required to solve the problem with a certain specified probability (say, 99%) (see [6]). Table 1 shows the computational effort for the ten approaches to solving this problem.

Table 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

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 use of multiple semi-isolated subpopulations. This result is consistent with the analysis of Sewell Wright [10] regarding demes of biological populations. Subpopulations can, of course, also be created and used on a serial machine.

6 Increasing performance: the PowerPC in the transputer architecture

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 Aachen, 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 along with several special add-on libraries and tools to handle the additional processor. 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.

Although our first implementation of parallel genetic programming had served us well for two years and was still functioning well in spring 1995, we needed greater computational power. Thus, we moved to a PowerPC based system. 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. This week included 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.

7 Using parallelism to achieve near-human performance on four problems

There have been several recent examples of problems (from the fields of cellular automata, molecular biology, and circuit design) in which genetic programming, running on our parallel PowerPC system, evolved a computer program that produced results that were slightly better than, or as good as, human performance for the same problem.

For example, various human-written algorithms have appeared in the past two decades for the majority classification task for one-dimensional two-state cellular automata. Parallel genetic programming with automatically defined functions has evolved a rule for this task with an accuracy of 82.326% [22]. This level of accuracy exceeds that of the original human-written Gacs-Kurdyumov-Levin (GKL) rule, all other known subsequent human-written rules, and all other known rules produced by automated approaches for this problem.

A second example involves the transmembrane segment identification problem where the goal is to classify a given protein segment (i.e., a subsequence of amino acid residues from a protein sequence) as being a transmembrane domain or non-transmembrane area of the protein (without using biochemical knowledge concerning hydrophobicity typically used by human-written algorithms for this task). Four different versions of parallel genetic programming have been applied to this problem [6][23][24]. The performance of all four versions evolved using genetic programming is slightly superior to that of algorithms written by knowledgeable human investigators.

A third example illustrates how automated parallel methods may prove to be useful in discovering biologically meaningful information hidden in the rapidly growing databases of DNA sequences and protein sequences. Parallel genetic programming successfully evolved motifs for detecting the D-E-A-D box family of proteins and for detecting the manganese superoxide dismutase family [25]. Both motifs were evolved without prespecifying their length, and both detect the two families either as well as, or slightly better than, the comparable human-written motifs found in the PROSITE database created by an international committee of experts on molecular biology.

Finally, our parallel system has been used to evolve both the topology and component values of electrical circuits to solve a variety of engineering problems, including low and high pass filters [26], multiple band pass filters [27], woofer and tweeter filters [28], and simple amplifiers [?]. Parallel genetic programming was able to evolve a solution for an asymmetric band-pass filter problem that was presented as a difficult-to-solve filter problem in an analog circuit journal [28].

Acknowledgments

Hugh Thomas of SGS-Thompson Microelectronics wrote configuration scripts and assisted in debugging the program and systems. We'd like to thank Transtech and Parsytec for providing technical assistance and support. Simon Handley and Forrest Bennett made numerous helpful comments on this article.

Bibliography

[1] 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.

[2] Tanese, R. (1989). *Distributed Genetic Algorithm for Function Optimization.* PhD. dissertation. Department of Electrical Engineering and Computer Science. University of Michigan.

[3] Goldberg, D. E. (l989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

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

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

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

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

[8] Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

[9] Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press.

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

[11] 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.

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

[13] 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.

[14] Singleton, A. (1994). Personal communication.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Andre, David, Bennett III, Forrest H, and Koza, John R. 1996. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press. In Press.

[23] Koza, John R. and Andre, David. 1996. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming II.* Cambridge, MA: MIT Press.

[24] Koza, John R. and Andre, David. 1996. Evolution of iteration in genetic programming. In *Evolutionary Programming V*: *Proceedings of the Fifth Annual Conference on Evolutionary Programming*. Cambridge, MA: MIT Press. In Press.

[25] Koza, John R. and Andre, David. 1996. Automatic discovery of protein motifs using genetic programming. In Yao, Xin (editor). 1996. *Evolutionary Computation: Theory and Applications*. Singapore: World Scientific.

[26] Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996c. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In Gero, John S. and Sudweeks, Fay (editors). *Artificial Intelligence in Design '96.* Dordrecht: Kluwer Academic Publishers.

[27] Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1996. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996:*

Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University. Cambridge, MA: The MIT Press.

[28] Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1996. Automated WYWIWYG design of both the topology and component values of analog electrical circuits using genetic programming. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: The MIT Press.

[29] Koza, John R., Andre, David, Bennett III, Forrest H, and Keane, Martin A. 1996. *Genetic Programming III*.. In Preparation.