Genetic Parallel Programming: Design and Implementation

Sin Man Cheang smcheang@vtc.edu.hk
Department of Computing, Hong Kong Institute of Vocational Education (Kwai Chung Campus), 20 Hing Shing Road, Kwai Chung, Hong Kong

Kwong Sak Leung ksleung@cse.cuhk.edu.hk
Kin Hong Lee khlee@cse.cuhk.edu.hk
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Sha Tin, Hong Kong

Abstract
This paper presents a novel Genetic Parallel Programming (GPP) paradigm for evolving parallel programs running on a Multi-Arithmetic-Logic-Unit (Multi-ALU) Processor (MAP). The MAP is a Multiple Instruction-streams, Multiple Data-streams (MIMD), general-purpose register machine that can be implemented on modern Very Large-Scale Integrated Circuits (VLSIs) in order to evaluate genetic programs at high speed. For human programmers, writing parallel programs is more difficult than writing sequential programs. However, experimental results show that GPP evolves parallel programs with less computational effort than that of their sequential counterparts. It creates a new approach to evolving a feasible problem solution in parallel program form and then serializes it into a sequential program if required. The effectiveness and efficiency of GPP are investigated using a suite of 14 well-studied benchmark problems. Experimental results show that GPP speeds up evolution substantially.

Keywords
Genetic programming, linear genetic programming, genetic parallel programming, parallel processor architecture.

1 Introduction

Genetic Programming (GP) (Koza, 1992; Banzhaf et al., 2000) is a branch of Evolutionary Computation that evolves computer programs. GP has been widely used in different application areas, such as medical data mining (Brameier and Banzhaf, 2001; Wong and Leung, 2000), speech recognition (Conrads et al., 1998), symbolic regression, recursive sequence regression (Huelsbergen, 1997), data classification (Kishore et al., 2000; Muni et al., 2004), analogy circuit design (Koza et al., 1999), combinational logic circuit design (Miller et al., 2000), automatic software re-engineering (Ryan, 2000), financial applications (Svangård et al., 2002), etc. Since the introduction of standard GP by Koza (1992) in the early 90's, numerous GP paradigms have been proposed by the community, which include linear structured GP (linear GP) (Banzhaf et al., 1998), graph-based GP (Poli, 1999; Teller and Veloso, 1996), stack-based GP (Perkis, 1994; Stoffel and Spector, 1996), Cartesian GP (Miller and Thomson, 2000), concurrent GP (Trenaman, 1999), grammar-based GP (Wong and Leung, 2000), etc. We shall describe some of these GP paradigms in Section 2. Most of them execute their genetic programs in sequential mode, either with a stack-based or recursive function calling mechanism. However, only a handful of GP researchers have investigated parallel program representations directly. Ryan and Ivan (2000) developed a software re-engineering system, *Paragen*, that uses standard GP to parallelize a sequential program to a parallel program. Practically speaking, parallelizing a sequential program does not usually produce an efficient parallel program for a specific architecture because *Paragen* only relies on the instructions in the sequential program.

The population-based nature of GP makes it easy to be parallelized with a suitable parallel architecture because the genetic programs in a population are independent. Multiple genetic programs can be evaluated in parallel in multiple evaluation engines (processors). In most cases, the speedup is linear (and even super-linear in some cases [Andre and Koza, 1997]) with respect to the number of processors. Theoretically, any GP paradigm can be sped up by parallelizing individual fitness evaluation with a parallel architecture, for example, an interconnected microcomputer LAN. Moore's law predicts that the speed of commercial computers continues to double approximately every 18 months (Moore, 1996). The continuous growth in computational power and network capacity directly benefits the parallel fitness evaluation. In the last decade, many distinct parallel GP (PGP) paradigms have been studied and proposed in the literature. Some elements of these models will be described in Section 2.

In this paper, we propose a novel linear GP paradigm, Genetic Parallel Programming (GPP) (Leung et al., 2002, 2002a). GPP evolves parallel programs that run on a tightly coupled, Multiple Instruction-streams, Multiple Data-streams (MIMD) register machine called a Multi-Arithmetic-Logic-Unit (Multi-ALU) Processor (MAP). The architecture of the MAP is simple and the hardware easy to implement (Lau et al., 2005).

Even though building a parallel processor in hardware is cost effective, writing an efficient parallel program to solve a problem on a specific parallel architecture is a difficult task for human programmers. It is prone to error and time consuming. The rationale behind the development of the GPP paradigm is to create parallel programs (algorithms) directly based on a parallel architecture in a reasonable computation time without human intervention.

So far, we have applied GPP to evolve parallel programs for different problems, which include numeric function regression, recursive sequence regression, Boolean function regression, artificial ant, data classification and combinational logic circuit design (Cheang et al., 2003, 2004; Leung et al., 2002, 2002a). Experimental results show that GPP can discover very compact parallel programs which a human programmer would not likely find. In order to investigate the evolutionary performance of GPP, we have performed a series of experiments using GPP to evolve programs for 14 benchmark problems in different areas. Experimental results show that parallel programs (for a MAP with more than one ALU) can be evolved more efficiently than sequential programs (for a MAP with one ALU). For example, in a Fibonacci recursive sequence regression experiment, evolving a 1-ALU sequential program requires, on average, 26.2 times more running time than an 8-ALU MAP parallel program. In some problems, super-linear speedups versus the number of ALUs are recorded. We have reported this speedup phenomenon in our previous papers (Cheang, 2003; Leung et al., 2003).

This paper is organized as follows: In Section 2, we outline the background and related work of this research. In Sections 3 and 4, we discuss the GPP paradigm and its advantages, respectively. We present the experiments in Section 5, and the results in Section 6. The summary and conclusions are given in Sections 7 and 8 respectively.

2 Background and Related Work

In this section, we briefly describe some popular GP paradigms which are significant in GP research.

2.1 Representations of Genetic Programs

There are two main streams in GP, standard GP (Koza, 1992) and linear structured GP (linear GP) [Banzhaf et al., 1998; Nordin, 1994]. In standard GP, a genetic program is represented in a tree structure, similar to the S-expression structure used in LISP. Genetic operations manipulate branches and leaf nodes of program trees, such as swapping two sub-trees and changing the function of a non-leaf node within a genetic program. The main advantage of standard GP is that it can maintain the correctness of the evolved offspring without syntax errors. However, due to the pointer-based representation of program trees in standard GP, the overheads of the program executions are relatively higher than those of Linear GP. This occurs because running a program tree on a classical Von Neumann machine involves stack operations, recursive function calls, and program tree interpretation. Pointer machines (Lee et al., 1990) can speed up program tree execution but it is not a cost-effective solution.

In linear GP, a genetic program is represented in a linear list of program instructions, either in machine languages (Heywood and Zincir-Heywood, 2000; Huelsbergen, 1997; Nordin, 1994; Nordin et al., 1999) or in high-level languages (Brameier and Banzhaf, 2001; O'Neill and Ryan, 1999). Linear GP is based on the principle of register machines and directly evolves program instructions that access program variables or registers. Genetic programs can run directly on a real machine without any additional translation process. All genetic evolutionary techniques spend a large portion of the processing time on the individual fitness evaluation. Direct instruction execution in linear GP, instead of program tree interpretation in standard GP, results in higher evolutionary performance. In linear GP, crossovers swap segments of codes between two individuals. Different crossover methods have been explored and exploited by the linear GP community. Nordin et al. proposed a block-based crossover that swaps 32-bit instruction blocks (Nordin et al., 1999a). Each block may contain any combination of 8-, 16-, 24- and 32-bit instructions. The motive of the block-based crossover is to improve the efficiency of crossover in Complex Instruction Set Computer (CISC) processors. Heywood and Zincir-Heywood (2000a) proposed a page-based linear GP (page-based GP) paradigm that performs crossover between pages. Each page contains a fixed number of instructions. Experimental results show that page-based GP is capable of providing concise solutions and is superior in terms of computational efforts (Heywood and Zincir-Heywood, 2002). Linear GP has been adopted for different real-life applications, such as medical data mining (Brameier and Banzhaf, 2001), recursive function regression (Huelsbergen, 1997), speech recognition (Conrads et al., 1998), financial instruments trading strategy learning (Svangård et al., 2002), and visual learning (Krawiec and Bhanu, 2003). Moreover, *Discipulus*$^{\text{TM}}$ (Francone, 2001), the first commercial linear GP system, evolves computer programs automatically and de-compiles linear program solutions into Java, C, and Intel assembly codes.

Besides standard and linear GP, various representations of genetic programs have been proposed. Perkis (1994) and Stoffel and Spector (1996) proposed stack-based GP systems in which genetic programs run on a stack-based virtual machine. In stack-based GP, a genetic program is represented in a string of symbols (functions and terminals) executed by the stack-based virtual machine. Experimental results show that stack-based GP systems require less computational effort than traditional S-expression-based GP systems.

Poli (1999) proposed a graph-based GP paradigm, known as Parallel Distributed GP (PDGP), in which genetic programs are represented by a directed graph. PDGP does not use genotype-phenotype mapping. It uses sophisticated crossover and mutation to manipulate sub-graphs. Teller and Veloso (1996) proposed another graph-based GP system, known as Parallel Algorithm Discovery and Orchestration.

Kantschik and Banzhaf (2001, 2002) proposed two hybrid GP representations, namely the linear-tree GP and the linear-graph GP. These two GP paradigms represent genetic programs in direct graph structures. Each node in a genetic program consists of a linear list of functional instructions followed by a branch instruction. The functional instructions perform simple operations to update a set of variables. The last branch instruction performs a test on variables to determine the next node to be executed. In order words, each node in a genetic program defines a macro-function and the tree or graph structure defines the control flows. Linear and point-based genetic operators are used at the node and the tree or graph levels respectively.

A few researchers have investigated parallel structures for genetic programs. Cartesian GP, developed by Miller et al. (Miller, 1999; Miller and Thomson, 2000; Walker and Miller, 2004), uses a two-dimensional array of functional units to represent genetic programs. A functional unit consists of both inputs and outputs. Each functional unit is labelled by a fixed sequence number. Outputs of a functional unit can be re-used as inputs of any functional units in its subsequent columns. A genetic program is represented by a list of multi-dimensional integer vectors. Each vector represents the input connections and the function of a functional unit. The final outputs are extracted from the outputs of any intermediate functional units. Cartesian GP is especially suitable for multi-output problems, for example, multi-level combinational logic circuits design (Miller et al., 2000). The main drawback of Cartesian GP is that the size of the phenotype must be fixed before an evolution can be run. Kalganova and Miller (1999) proposed an approach to vary the geometry of the two-dimensional array during evolution. Both the numbers of columns and rows in the array can be altered during run-time. However, complicated pre-evaluation correction processes are needed to maintain valid genetic programs.

Concurrent GP developed by Trenaman (1999) represents a genetic program in multiple program-trees. These program-trees are executed in asynchronous (interleaved) schemes. Concurrent GP is designed for controlling an agent (a machine that acts intelligently in its environment). The collective group behaviour of all program-trees becomes the behaviour of the agent. The experimental results of the virtual robotic problems show that concurrent GP can produce better agents than conventional single-tree GP. The main drawback of concurrent GP is that it has only tackled agent problems so far.

2.2 Parallelization of GP

GP paradigms can be parallelized in four main models: 1) the global model; 2) the island model; 3) the fine-grained model; and 4) the cooperative co-evolution model.

In the global model, genetic programs are evaluated in parallel by different processors. Usually, a master process assigns genetic programs to different slave processors. During the evaluation, there is no communication between processors. This model is easy to implement but it may suffer from load imbalance due to the different execution complexities of genetic programs (Tomassini, 1999).

In the island model (Andre and Koza, 1996; Brameier et al., 1999; Koza, 2003; Martin et al., 1997), genetic programs in a population are equally divided into a number of sub-populations (demes). A standard GP algorithm works on each deme. Demes communicate according to a communication topology and exchange genetic programs periodically by migrating genetic programs amongst demes.

In the fine-grained model (Folino et al., 2003; Juille and Pollack, 1995), each genetic program is associated with a cell location on a multi-dimensional grid. Each cell can only interact with its direct neighbours. Fitness evaluation is performed simultaneously on all cells. Genetic information may diffuse slowly across the grid. The fine-grained model is suitable to be implemented on massively parallel hardware architecture.

In the cooperative co-evolution model (Krawiec and Bhanu, 2003; Potter, 1997; Potter and De Jong, 2000), the solution of a problem is divided into sub-components. Each of these sub-components is evolved in a genetically isolated sub-population (species). These species collaborate with one another to achieve a top level goal of finding an optimal solution for the problem. A complete solution is obtained by assembling representatives (best individual sub-components) from each of the species. The evolution of genetically isolated species in separated sub-populations can be easily distributed across a coarse-grained parallel architecture (such as a cluster of microcomputers connected by a LAN) with little overhead. Experimental results show that this model improves the evolution speed of both GA and GP significantly.

All of the above parallelization models speed up evolution by multiple evaluation engines. We classify this higher-level parallelism as parallel GP (PGP). However, in this paper, we do not work on PGP but on the evolution of genetic programs represented in a specific parallel form.

3 Genetic Parallel Programming

Genetic Parallel Programming (GPP) is a novel linear GP paradigm that evolves parallel programs of an MIMD architecture with multiple arithmetic-logic-units (ALU). Figure 1 shows the framework of a GPP system.

It consists of two core components, a Multi-ALU Processor (MAP) and an Evolution Engine (EE). The MAP is an execution engine for genetic program fitness evaluation. The EE manipulates the genetic program population, performs genetic operations (such as mutation, crossover, etc.) and de-compiles the solution program into symbolic assembly and high-level language codes. The following subsections provide details of GPP.

3.1 Multi-ALU Processor Architecture

In a GPP system, the configurations (including function sets, data type, etc.) of its MAP are problem dependent. Figure 1 shows a block diagram of a MAP with w ALUs, p ports, v variable registers (reg[0] to reg[$v-1$]) and c constant registers (reg[v] to reg[$v+c-1$]). The w ALUs access a shared register-file through a programmable crossbar switching-network and p ports. Each port can read any one variable or constant register. Thus, most p different register values can be transferred to the crossbar switching-network simultaneously. A port can be shared by multiple ALU inputs. In addition, each ALU maintains two status-flags (Zero and Negative flags). There are, in total, $2w$ flags (2 flags $\times$ w ALUs) which are used to determine conditional branches. All intermediate and output values are stored in the variable registers. In order to prevent multiple ALUs from writing to the same register simultaneously, each ALU is restricted to writing to an exclusive subset of variable registers. The v variable registers are equally distributed to the w ALUs as output registers, resulting in v/w registers for each ALU. For example, as shown in Figure 1, alu[i] can write its output to any one of the variable registers from reg[iv/w] to reg[$(i+1)v/w-1$]. The constant registers store constants and input values. These constant registers are read-only and their values are initialized and manipulated by the EE. The ALUs cannot write to any constant registers. Fixed constants (e.g. 0, 1, 3.14, etc.) are loaded into the MAP before a program is executed and they cannot be modified afterwards. Random constants can be changed by a constant mutation operator as described in Section 3.3.

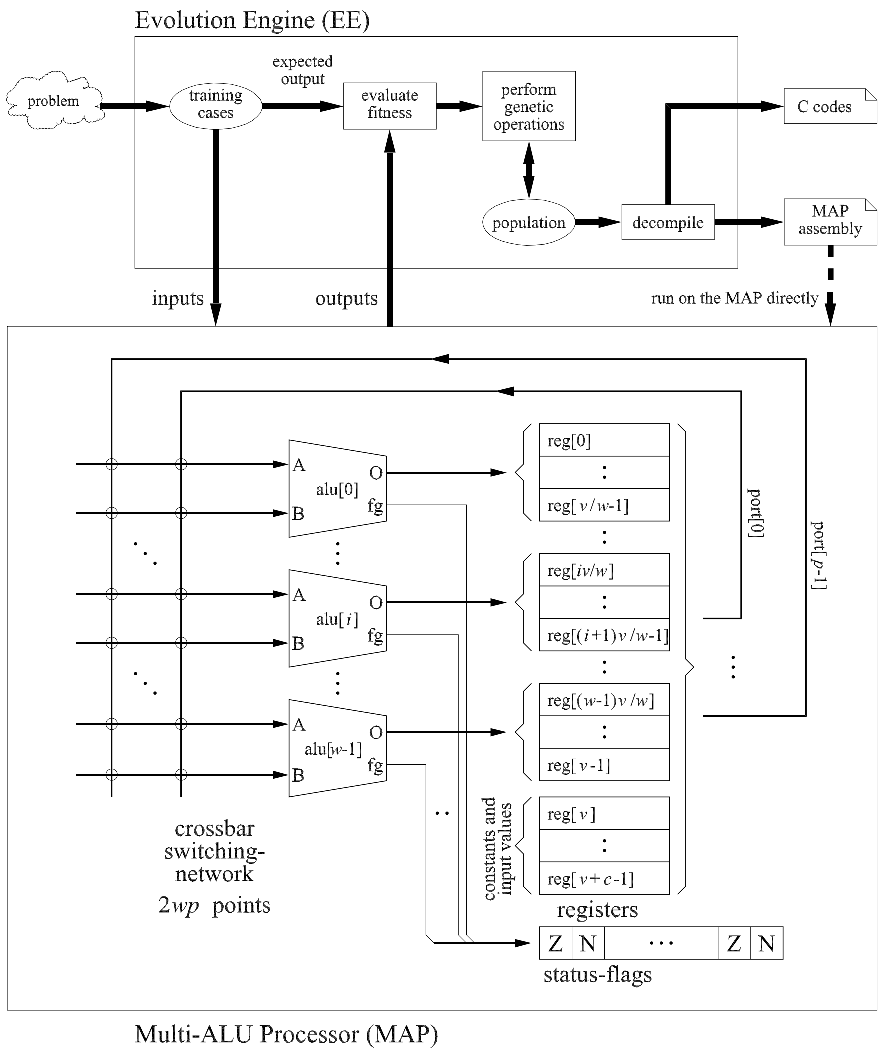


Figure 1: The GPP system framework.

In a MAP, the number of ALUs (w) and ports (p) are the two most important factors which affect the parallelism of the evolved genetic programs. w defines the maximum number of parallel operations which can be executed in each clock cycle. p defines the maximum number of different values that can be transferred from registers to ALUs in each parallel-instruction. Too many ports will consume unnecessary hardware resources but too few ports will hamper the degree of parallelism.

3.2 Instruction Format

The genotype of a genetic program is a sequence of control-codes and the correspondent phenotype is a parallel assembly program (a MAP program) with a sequence of parallel-instructions. A parallel-instruction consists of w sub-instructions that perform w operations simultaneously in each clock cycle. Figure 2 shows the syntax of a MAP parallel-instruction ($\langle$pi$\rangle$) which consists of two main parts, a branch sub-instruction ($\langle$bsi$\rangle$) and w ALU sub-instructions ($\langle$asi[0]$\rangle$ to $\langle$asi[$w-1$]$\rangle$).

$\langle$pi$\rangle$	::=	**addr :** $\langle$bsi$\rangle$, $\langle$asi[0]$\rangle$, ... , $\langle$asi[i]$\rangle$, ... , $\langle$asi[$w-1$]$\rangle$
$\langle$bsi$\rangle$	::=	$\langle$bop$\rangle$ $\langle$balu$\rangle$ **boffset**
$\langle$bop$\rangle$	::=	**nxt**|**end**|**jmp**|**jgt**|**jlt**|**jep**|**jne**|**jge**|**jle**|...
$\langle$balu$\rangle$	::=	**0** | ... | $w-1$
$\langle$asi[i]$\rangle$	::=	$\langle$aop[i]$\rangle$ $\langle$ireg$\rangle$ [$\langle$ireg$\rangle$] $\langle$oreg[i]$\rangle$
$\langle$aop[i]$\rangle$	::=	**add**|**sub**|**mul**|**div**|**mov**|**inc**|**dec**|**not**|**and**|**or**|**xor**|**nop**|...
$\langle$ireg$\rangle$	::=	**r00**|...|**r**$(v+c-1)$
$\langle$oreg[i]$\rangle$	::=	**r**(iv/w) | ... | **r**$((i+1)v/w-1)$

Figure 2: The syntax of a MAP parallel-instruction.

address	$\langle$bsi$\rangle$	$\langle$asi[0]$\rangle$	$\langle$asi[1]$\rangle$	$\langle$asi[2]$\rangle$	$\langle$asi[3]$\rangle$
00:	jgt alu0 0,	dec r0 r0,	inc r14 r5,	inc r14 r8,	add r5 r14 r14
01:	end,	nop,	nop,	nop,	nop

Figure 3: A 4-ALU MAP program acquired by a GPP system to calculate the Fibonacci recursive sequence.

Figure 3 shows a 4-ALU MAP program acquired by a GPP system that calculates the Fibonacci sequence. It consists of one branch sub-instruction ($\langle$bsi$\rangle$) and four ALU sub-instructions ($\langle$asi[0]$\rangle$ to $\langle$asi[3]$\rangle$). In the first parallel-instruction, the branch sub-instruction *jgt alu0 0* directs the program to jump to PC+0 if the result of the last arithmetic operation token in the alu[0] is greater than zero. The remaining four ALU sub-instructions perform four arithmetic operations by the four ALUs:

$\langle$asi[0]$\rangle$: *dec r0 r0* controls alu[0] to decrease *r0* by 1;
$\langle$asi[1]$\rangle$: *inc r14 r5* controls alu[1] to add 1 to *r14* and write the result to *r5*;
$\langle$asi[2]$\rangle$: *inc r14 r8* controls alu[2] to add 1 to *r14* and write the result to *r8*; and
$\langle$asi[3]$\rangle$: *add r5 r14 r14* controls alu[3] to add *r5* to *r14*.

Noticeably, $\langle$asi[1]$\rangle$, $\langle$asi[2]$\rangle$, and $\langle$asi[3]$\rangle$ share *r14* as an input in the parallel-instruction. Both *r5* and *r14* are modified simultaneously. The second parallel-instruction stops the program with an end branch sub-instruction and four *nop* (no operation) sub-instructions. The port control codes are hidden and automatically determined by the ALU sub-instructions.

Figure 4 shows the control-codes of a parallel-instruction. *brh_ctrl.func* encodes a branch function, *brh_ctrl.alu* identifies an ALU whose status-flags are used to determine the branch condition and *brh_ctrl.offset* defines a 2's complement branch offset address. *alu_ctrl*[*].*func* encodes an ALU function, *alu_ctrl*[*].*ina* and *alu_ctrl*[*].*inb* encode ports connected to two inputs and *alu_ctrl*[*].*out* identifies an output register. Each *prt_ctrl*[*] encodes a register that gives value to the port. The encoding scheme allows an arbitrary bit pattern to become a valid control-code without causing run-time fatal errors. Thus, no pre-evaluation repairing process is needed. This closure property is especially important for GPP because of its random nature.

control	*brh_ctrl*			*alu_ctrl*[0]				...	*alu_ctrl*[$w-1$]				*prt_ctrl*[0]	...	*prt_ctrl*[$p-1$]
-codes	*func*	*alu*	*offset*	*func*	*ina*	*inb*	*out*	...	*func*	*ina*	*inb*	*out*		...	

Figure 4: The control-code of a parallel-instruction.

In this paper, we fix some MAP configurations, i.e. 8 branch functions (3-bit *brh_ctrl.func*), 32 branch addresses (5-bit *brh_ctrl.offset*) and 16 ALU functions (4-bit *alu_ctrl*[*].*func*). We can identify a MAP, $\mathcal{M}(w, p, v, c)$, with four parameters: w (the number of ALUs), p (the number of ports), v (the number of variable registers) and c (the number of constant registers). For example, a 1-ALU MAP ($w = 1$,$p = 2$) with eight variable ($v = 8$) and eight constant ($c = 8$) registers is denoted by $\mathcal{M}$(1,2,8,8).

3.3 Genetic Operators

The Evolution Engine (EE) manipulates the population, performs genetic operations, loads genetic programs to the MAP for fitness evaluations, calculates/reports statistics and de-compiles the solution parallel program to symbolic parallel assembly and high-level language codes. The five main genetic operators used in the EE are as follows:

Selection. The EE uses tournament selection to produce offspring.

Parallel-Instruction Level Crossover. In each tournament, two parents are selected from the tournament set. In each parent, two randomly selected parallel-instruction addresses are used to determine a segment of the parallel-instructions for crossover. Since the crossover is conducted at parallel-instruction level, all sub-instructions in each individual parallel-instruction are kept. It retains the cooperative power of sub-instructions in each individual parallel-instruction.

Bit Mutation. This randomly flips bits in a genotype, based on a fixed probability.

Constant Mutation. Based on a fixed probability, the value of a constant register is replaced by a random number (with a uniform distribution) between two predefined boundary values.

Diversity Maintenance. In order to maintain the diversity of the population, the EE adopts an individual replacement technique similar to pre-selection (Mahfoud, 1992). In each tournament, two children are evolved and evaluated. Then, the better child will be selected and compared with its two parents. If its fitness is different from both of its parents, it will replace the worst individual in the tournament set. Otherwise, it will be discarded. This operator avoids similar individuals filling up the population and hence increases the diversity of the search.

3.4 De-compilation and Serialization

We can use a GPP system to evolve a parallel program solution and then serialize it to a sequential one. The sequential program can then be ported to other systems. The one-time serialization is a deterministic process with the linear order of computational complexity with respect to the size of the original MAP program. Currently, the GPP system can produce an evolved solution in two formats, a MAP program and a C code segment. The MAP program can be assembled into a sequence of MAP machine codes to run on a MAP directly. The C code segment can be embedded into other C programs.

4 Advantages of GPP

The advantages of GPP are discussed below.

4.1 Reducing Usage of Intermediate Registers

Since a MAP consists of multiple data paths and allows multiple registers to be updated in a clock cycle, multiple results can be written to different variable registers simultaneously. Thus, the demand for intermediate registers and the complexity of the solution programs can be reduced. We can show this advantage by a simple example, the rotation of values in three registers. In Figure 5(a), a 1-ALU MAP program uses one temporary register (t) and four sequential *move* operations. In Figure 5(b), a 2-ALU MAP program uses one temporary register and two parallel-instructions. Each parallel-instruction consists of two *move* operations. In Figure 5(c), a 3-ALU MAP program uses only one parallel-instruction (without temporary register) and three *move* operations. This example shows that the specific architecture of the MAP can reduce the usage of intermediate registers and operations.

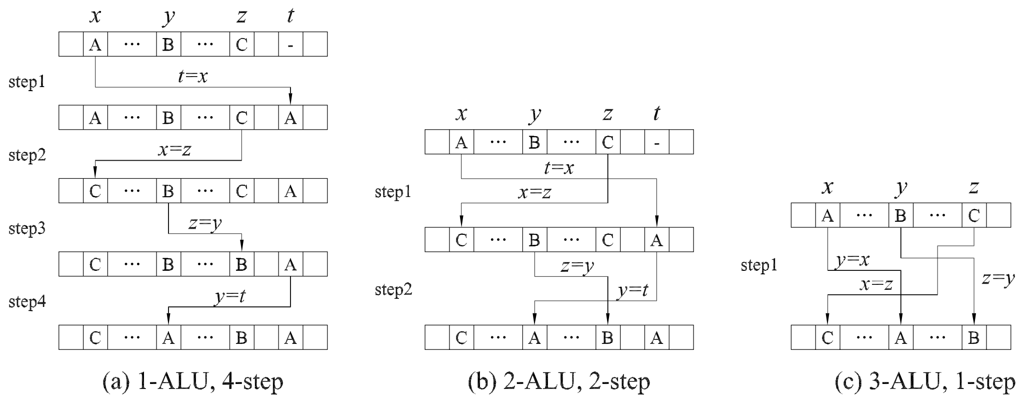


Figure 5: Three MAP programs (with different numbers of ALUs) which rotate values in three registers.

4.2 Evolving Programs with GPP

We can still run sequential programs on a multi-ALU MAP by using one ALU only in each parallel-instruction and letting all the remaining ALUs be *nop*. However, writing an optimal parallel program that can fully make use of the parallelism of the MAP is a difficult task. It involves four sub-tasks: 1) deriving a sequential algorithm to solve

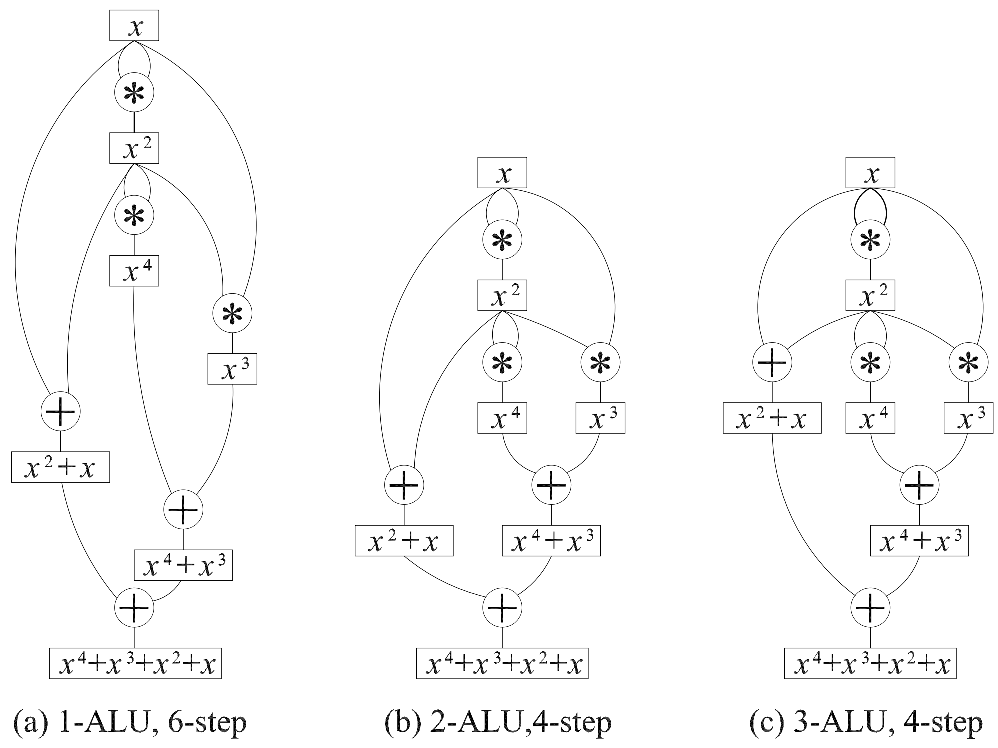


Figure 6: Three equivalent parallel flow diagrams to calculate x^4+x^3+x^2+x.

the problem; 2) determining dependencies of instructions; 3) resolving processor resource constraints, such as the limited numbers of ports and ALUs; and 4) allocating sub-instructions to parallel-instructions. The first sub-task is heuristic-based and the third sub-task is NP-complete. The development of an optimal parallel program is an error prone and time-consuming task. This provides the first indication that evolving a parallel program is more difficult and complicated than evolving a sequential program. Nevertheless, experimental results contradict our conjecture. After performing a series of experiments on some benchmark problems on different MAPs with different w (numbers of ALUs), it is observed that GPP can evolve wide programs (with a large w) more efficiently than narrow programs (with a small w) (Leung et al., 2003).

In Sections 5 and 6, we shall present 14 experiments, and their results, to illustrate this speedup phenomenon. The underlying principle behind the phenomenon is that the more the number of parallel ALUs is used, the higher the probability of an expected operation will be hit for each step (sequential- or parallel-instruction) assuming operations within a step are independent. Similar to flipping coins, if we flip one coin, the probability of coming up with a ″head″ (expected outcome) is 0.5. If we flip two coins, the probability of coming up with ″head(s)″ increases to 0.75 $(1 - 0.5 \times 0.5)$. Say, if we need at least one ″head″ in two consecutive trials given four coin chances, we can either flip two coins in each trial, or one coin in each trial, and repeat. The respective probabilities are 0.5265 (0.75×0.75) and 0.4375 $(1 - 0.75 \times 0.75)$, favouring the former approach.

P[parallel-instruction crossover]	1.0	population size	2000
P[constant crossover]	0.5	tournament size	10
P[bit mutation]	0.02	independent runs	100/MAP/problem
P[constant mutation]	0.01		

Table 1: Common experimental settings.

symbolic names	descriptions
ALU functions (F)	
add, *sub*, *mul*, *div*, *inc*, *dec*	arithmetic: +, $-$, $\times$, %, ++, $--$
mov	copy a register's value to other register
and, *or*, *nan*, *nor*	logic: AND, OR, NAND, NOR
and, *xor*, *shr*	bit-wise: AND, XOR, right-shift 1-bit
nop	no operation
lft, *rgt*, *wlk*	artificial ant: copy 'L' (*turn_left*), 'R' (*turn_right*), and 'W' (*go_ahead*) to the output register
branch functions (B)	
jmp	unconditional: jump
jeq, *jgt*, *jlt*	conditional: jump equal, jump greater than, jump less than
nxt	advance the program counter by one
end	terminate execution after the instruction

Table 2: All branch and ALU functions used in experiments.

This phenomenon opens up a new approach to evolving a sequential program in two stages: 1) evolving a satisfactory parallel program with reduced computational effort; and 2) serializing the evolved parallel program into a sequential program. Obviously, the evolution stage takes up a large portion of the processing time. Consequently, the evolutionary process based on linear GP can be sped up.

4.3 Separating Program Flow from ALU Operations

In standard linear GP, an instruction performs either an ALU or a branch operation. A mutation on the op-code of an instruction may alter an ALU operation to a branch operation and vice versa. Such mutation changes the phenotype of the genetic programs dramatically. In GPP, the branch and ALU operations are encoded explicitly in different parts of a genotype. So, a branch operation is always mutated to a branch operation while an ALU operation is always mutated to an ALU operation.

4.4 Discovering Parallel Algorithms

We can obtain different parallel algorithms automatically by GPP using MAPs with different w. For example, the polynomial $x^4 + x^3 + x^2 + x$ can be calculated by different algorithms, either in sequence or parallel. Three equivalent data flow diagrams are shown in Figure 6. Figure 6(a) shows that a 1-ALU MAP takes six steps to calculate the result. Figures 6(b) and (c) show the executions of the same programs with 2- and 3-ALU MAPs respectively. Both of them take only four steps to calculate results.

function sets	F={*add*, *sub*, *mul*, *div*, *mov*, *nop*}, B={*nxt*, *end*}
MAPs	$\mathcal{M}$(*,*,8,8)
terminal set	r15$\leftarrow x$, r4$\rightarrow y$
random constants	floating-point values $\in[-1.0,1.0]$
training samples	CUB, SEX : 50 values of $x \in[-1.0,1.0]$; 3PI : 10 values of $x \in[0.5,10.0]$; 3SI : 50 values of $x \in[-\pi,\pi]$
raw fitness (f_{raw})	average absolute error of y
success predicate	CUB, SEX : $f_{raw} \leq 0.01$; 3PI : $f_{raw} \leq 0.05$; 3SI : $f_{raw} \leq 0.26$
data type	floating-point
evaluation time	maximum l clock cycles (each program)
termination	10^7 tournaments or no improvement over 10^6 tournaments

Table 3: Settings for the four single-input numeric function experiments (CUB, SEX, 3PI, and 3SI).

function sets	F={*add*, *sub*, *mul*, *div*, *mov*, *nop*}, B={*nxt*, *end*}
MAPs	$\mathcal{M}$(*,*,8,8)
terminal set	r10$\leftarrow l_1$, r11$\leftarrow w_1$, r12$\leftarrow h_1$, r13$\leftarrow l_2$, r14$\leftarrow w_2$, r15$\leftarrow h_2$, r4$\rightarrow y$
random constants	integer values 0 or 1
training samples	10 samples of input values $l_i, w_i, h_i \in[1,50]$
raw fitness (f_{raw})	total absolute error of y
success predicate	f_{raw}=0
data type	32-bit signed integer
evaluation time	maximum l clock cycles (each program)
termination	10^7 tournaments

Table 4: Settings for the Two-box (2BX) experiment.

5 Experiments

In order to investigate the GPP paradigm, a series of experiments on 14 benchmark problems (including numeric functions, Boolean functions, artificial ant, and data classifications) were carried out. The experimental results are presented in Section 6. These problems were tested with four different numbers of ALUs (w = 1, 2, 4 and 8) and five different numbers of maximum program lengths (l = 8, 16, 32, 64 and 128 parallel-instructions). Thus, a total of 20 different MAP configurations were tested for each problem. We developed the software emulators of all the MAPs using Microsoft Visual C. All experiments were run on 3.0GHz Intel Pentium 4 PCs with Microsoft Windows XP. Tables 1 and 2 detail the common experimental settings and the functions (both ALU and branch) used in the experiments respectively.

5.1 Descriptions of Benchmark Problems

Single-input numeric functions. There are four floating-point function regression problems as follows:

1. Cubic function (CUB) : $y = (x+1)^3, x \in [-1.0, 1.0]$;
2. Sextic function (SEX) : $y = x^6 - 2x^4 + x^2, x \in [-1.0, 1.0]$;
3. 3-π function (3PI) : $y = (x/\pi) + (x^2/\pi) + 2x\pi, x \in [0.5, 10.0]$; and

function sets	F={*and*, *or*, *nan*, *nor*, *nop*}, B={*nxt*, *end*}
MAPs	$\mathcal{M}$(*,*,8,8)
terminal set	3EP : r8-r10$\leftarrow x_1$-x_3, r4$\rightarrow y$; 4EP : r8-r11$\leftarrow x_1$-x_4, r4$\rightarrow y$; 5EP, 5SM : r8-r12$\leftarrow x_1$-x_5, r4$\rightarrow y$
random constants	Boolean values 0 or 1
training samples	2^n truth table rows
raw fitness (f_{raw})	ratio of rows not satisfied
success predicate	f_{raw}=0
data type	Boolean
evaluation time	maximum l clock cycles (each program)
termination	10^8 tournaments or no improvement over 10^7 tournaments

Table 5: Settings for the four Boolean experiments (3EP, 4EP, 5EP, and 5SM).

function sets	F={*add*, *shr*, *inc*, *dec*, *mov*, *nop*}, B={*jmp*, *jeq*, *jgt*, *nxt*, *end*}
MAPs	$\mathcal{M}$(*,*,8,8)
terminal set	r15$\leftarrow i$, r4$\rightarrow T_i$;
random constants	integer values 0 or 1
training samples	10 integer values of i $\in$[0,24]
raw fitness (f_{raw})	total relative error of T_i
success predicate	f_{raw}=0
data type	32-bit integer
evaluation time	maximum 200 clock cycles (each program)
termination	10^8 tournaments or no improvement over 10^7 tournaments

Table 6: Settings for the Fibonacci (FIB) experiment.

4. 3-sine function (3SI) : $y = \sin x + \sin 2x + \sin 3x, x \in [-\pi, \pi]$.

The objective of these four experiments is to evolve programs that approximate the target functions within preset average absolute error values. Table 3 shows the settings for the four experiments.

Multi-input numeric function. This is a multi-input function regression problem that calculates the difference between the volumes of two rectangular boxes (2BX), $y = l_1w_1h_1 - l_2w_2h_2$. The l_i, w_i, and h_i represent the length, width, and height of a box respectively. The objective is to minimize the total absolute error of y to as close to zero as possible. Table 4 shows the settings for this experiment.

Boolean functions. There are four Boolean functions as follows:

1. 3-even-parity (3EP) : $y = x_1 \oplus x_2 \oplus x_3$;
2. 4-even-parity (4EP) : $y = x_1 \oplus x_2 \oplus x_3 \oplus x_4$;
3. 5-even-parity (5EP) : $y = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$; and
4. 5-symmetry (5SM) : $y = \overline{(x_1 \oplus x_5)} \cdot \overline{(x_2 \oplus x_4)}$;

where $\oplus$ denotes an exclusive-OR (XOR) operation.

function sets	F={*and*, *xor*, *shr*, *mov*, *nop*}, B={*jmp*, *jeq*, *jgt*, *nxt*, *end*}
MAPs	$\mathcal{M}$(*,*,8,8)
terminal set	r15← i, r4→ p;
random constants	integer values -1, 0 and 1
training samples	54 32-bit integers together with their parity bit
raw fitness (f_{raw})	ratio of incorrect samples
success predicate	f_{raw}=0
data type	32-bit integer
evaluation time	maximum 200 clock cycles (each program)
termination	10^8 tournaments or no improvement over 10^7 tournaments

Table 7: Settings for the 32-bit even-parity (EPT) experiment.

function sets	F={*sub*, *mov*, *lft*, *rgt*, *wlk*, *nop*}, B={*jeq*, *nxt*, *end*}
MAPs	$\mathcal{M}$(*,*,8,8)
terminal set	r0← 1(*food_ahead*), 0(*no_food_ahead*),
	r4→ 'L'(*turn_left*), 'R'(*turn_right*), 'W'(*go_ahead*)
random constants	all set to zero
training samples	89 food pellets on a 32×32 toroid
raw fitness (f_{raw})	number of food pellets remained
success predicate	$f_{raw} = 0$ (eat all pellets)
data type	32-bit integer
evaluation time	- maximum 600 ant actions on the toroid; and
	- maximum 64 clock cycles per action
termination	10^8 tournaments or no improvement over 10^7 tournaments

Table 8: Settings for the Artificial ant - Santa Fe Trail (ANT) experiment.

The complete truth tables of these Boolean functions are used as training samples. A satisfactory solution program must solve all rows in a truth table. Table 5 shows the settings for the four experiments.

Fibonacci sequence (FIB). This is a recursive sequence, i.e. $T_0 = T_1 = 1$ and $T_i = T_{i-1} + T_{i-2}$. A solution program of this problem is very difficult to evolve because it includes loops. Table 6 shows the settings for this experiment. Three branch functions (*jmp*, *jep* and *jgt*) are added to the branch function set (B). We also relaxed the maximum individual evaluation time to 200 MAP clock cycles for program loops evolution.

32-bit even-parity (EPT). This accepts a 32-bit integer number and returns an even-parity bit of the 32 bits. This is not the same as the previous n-even-parity problems. Since all bits are stored as an integer number, a solution program should contain a loop with a *shr* (right-shift 1-bit) operation to extract bits from the input integer. Table 7 shows the settings for this experiment.

Artificial ant – Santa Fe Trail (ANT). The objective of this experiment is to determine a path for an artificial ant to eat all 89 food pellets lying along an irregular winding trail on a 32×32 cell toroid (Koza, 1992). A solution program should embed a Finite State Machine with two input values (*food_ahead* and *no_food_ahead*) from the front

function sets	F={*add*, *sub*, *mul*, *div*, *mov*, *nop*}, B={*jlt*, *jgt*, *jeq*, *nxt*, *end*}
MAPs	$\mathcal{M}$(*,*,16,16)
terminal set	BCW : r23-r31← a_1-a_9, r8→ c; BLD : r26-r31← a_1-a_6, r8→ c
random constants	floating-point values ∈[-1.0,1.0]
training samples	BCW : c1:c2=444:239 records; BLD : c1:c2=145:200 records
raw fitness (f_{raw})	number of mis-classified records
success predicate	BCW: $f_{raw} < 23$; BLD: $f_{raw} < 103$
data type	floating-point
evaluation time	maximum l clock cycles (each program)
termination	10^7 tournaments

Table 9: Settings for the two data classification experiments (BCW and BLD). In order to handle more input and constant values, we use MAPs with 16 variable and 16 constant registers to solve these two problems.

sensor of the ant, and three output values, including *turn_left* ('L'), *turn_right* ('R') and *go_ahead* ('W'). Table 8 shows the settings for this experiment. Three constant assignment functions (*lft*, *rgt* and *wlk*) are included in the ALU function set (F) to control the ant. For this problem, a solution program needs to be called repeatedly to guide the ant to move and eat food pellets on the toroid step-by-step. For each call, the program obtains the front sensors input through *r0*, and then determines the next action based on the current input and the previous state (values stored in variable registers).

Data classification. The objectives of these experiments are to learn classification programs for two preprocessed (Lim et al., 2000) binary-class UCI repositories of Machine Learning Databases (Merz and Murphy, 1996): 1) Wisconsin Breast Cancer (BCW); and 2) Bupa Liver Disorders (BLD). The numbers of input attributes of the BCW and BLD databases are nine (a_1-a_9) and six (a_1-a_6) respectively. Since the main purpose of this paper is to demonstrate the effectiveness of GPP, we have not fine-tuned the GP parameters for each database. All configurations are set as consistently as possible. As shown in Table 9, input attributes are stored in the upper portion of the constant registers and the result (c) is returned by *r8*.

6 Results and Evaluations

The main assertion of this paper is that 2-, 4-, and 8-ALU MAPs require less computational efforts to evolve solution programs than 1-ALU MAPs. We performed a Student's *t* test on the computational efforts of different MAPs. For each of the 14 problems, there are five different program length values (l = 8, 16, 32, 64, 128). For each l value, there are three *t* tests (i.e. 1- versus 2-ALU, 1- versus 4-ALU, and 1- versus 8-ALU). The computational efforts of multi-ALU MAPs are significantly lower than those of 1-ALU MAPs (all p-values are less than the 0.02 level of significance).

6.1 Evolutionary Speed

In order to compare the computational effort of different GPP systems with different MAP configurations, we adopt the computational effort (*E*) measurement method proposed by Koza (1992). The pure software emulator wall-clock execution time (evolution time) is estimated by

$$T = E \times \frac{\text{total execution time of all success runs}}{\text{total tournaments spent on all success runs}} \tag{1}$$

We performed 100 independent runs on each of the 14 problems with the 20 different combinations of w and l. To compare results of problems with different difficulties, relative values of E and T are used in this paper and calculated by

$$RelE_{w,l} = E_{w,l} \div \min_{\forall w,l}(E_{w,l}) \tag{2}$$

$$RelT_{w,l} = T_{w,l} \div \min_{\forall w,l}(T_{w,l}) \tag{3}$$

In Equations (2) and (3), $E_{w,l}$ and $T_{w,l}$ denote the values of E and T of a specific MAP with w ALUs and maximum l parallel-instructions respectively. The $RelE_{w,l}$ and $RelT_{w,l}$ of the 14 benchmark problems are listed in Table 10. In addition to the values of $RelE_{w,l}$ and $RelT_{w,l}$, the table shows four sets of average values:

$$AvgEA_w = \left[\sum_{\forall l}(RelE_{w,l}/RelE_{8,l}) \right] \div 5 \tag{4}$$

$$AvgTA_w = \left[\sum_{\forall l}(RelT_{w,l}/RelT_{8,l}) \right] \div 5 \tag{5}$$

$$AvgEI_l = \left[\sum_{\forall w}(RelE_{w,l}/RelE_{w,128}) \right] \div 4 \tag{6}$$

$$AvgTI_l = \left[\sum_{\forall w}(RelT_{w,l}/RelT_{w,128}) \right] \div 4 \tag{7}$$

In Table 10, a small value (e.g. 1.0) represents a low computational effort (high performance) while a large value represents a high computational effort (low performance). For example, the value of $AvgEA_4$ of the CUB experiment (1.3) is lower than that of $AvgEA_1$ (3.8). This means that GPP is more efficient in evolving a 4-ALU parallel program than evolving a 1-ALU sequential program for the CUB problem.

In some problems, such as the 4EP problem, GPP did not evolve any satisfactory programs in all 100 runs with specific MAP configurations. In these cases, E cannot be calculated because GPP cannot solve the problem or may need more computational effort to evolve a satisfactory solution program with the specific MAP configurations. For quantitative analysis, we fill the corresponding cell in Table 10 with the largest obtained values from other solvable MAP configurations. For example, as the results of the 4EP experiment show in Table 10, the cell of $RelE_{1,8}$ is filled with $>$649.8 (=$RelE_{1,16}$) which is the largest $RelE$ of all other tested MAPs with at least one successful run. The '$>$' indicates that the actual value is greater than the printed value. In addition to the numerical values shown in Table 10, we show graphical representations of the $RelE_{w,l}$ and $RelT_{w,l}$ of the 14 problems in Figures 7 and 8 respectively.

6.2 Individual Experimental Results

We would like to emphasize that when $w = 1$, GPP is actually evolving sequential programs. It is equivalent to the standard linear GP. Based on this proposition, we compare a multi-ALU MAP to a 1-ALU MAP directly.

	Relative Computational Effort ($RelE$)							Relative Evolution Time ($RelT$)						
		l=8	l=16	l=32	l=64	l=128	$AvgEA_w$		l=8	l=16	l=32	l=64	l=128	$AvgTA_w$
CUB	w=1	7.1	4.4	2.7	3.7	$\underline{4.8}$	3.8	w=1	3.3	3.1	2.3	4.0	$\underline{4.8}$	2.7
	w=2	2.2	2.0	1.6	$\underline{2.1}$	1.8	1.7	w=2	1.0	1.3	1.2	$\underline{1.7}$	1.8	1.1
	w=4	1.7	1.4	$\underline{1.3}$	1.3	1.7	1.3	w=4	1.0	1.2	$\underline{1.3}$	1.2	2.2	1.0
	w=8	1.2	$\underline{1.3}$	1.0	1.1	1.3	1.0	w=8	1.0	$\underline{1.4}$	1.1	1.3	1.8	1.0
	$AvgEI_l$	1.2	1.0	0.7	0.9	1.0		$AvgTI_l$	0.6	0.7	0.6	0.8	1.0	
SEX	w=1	64.4	15.9	7.7	5.8	$\underline{5.1}$	15.6	w=1	25.1	9.1	5.5	5.0	$\underline{4.5}$	8.7
	w=2	16.9	3.9	2.0	$\underline{1.8}$	1.7	4.2	w=2	6.6	2.2	1.5	$\underline{1.6}$	1.5	2.4
	w=4	4.0	1.4	$\underline{1.2}$	1.1	1.0	1.4	w=4	2.0	1.1	$\underline{1.3}$	1.1	1.0	1.1
	w=8	1.4	$\underline{1.1}$	1.1	1.1	1.0	1.0	w=8	1.0	$\underline{1.2}$	1.3	1.4	1.4	1.0
	$AvgEI_l$	7.0	2.0	1.2	1.1	1.0		$AvgTI_l$	3.2	1.4	1.1	1.1	1.0	
3PI	w=1	70.2	27.5	12.0	10.4	$\underline{8.0}$	17.9	w=1	21.8	12.5	7.9	8.1	$\underline{6.8}$	10.5
	w=2	14.6	10.4	3.8	$\underline{3.0}$	2.6	5.0	w=2	5.3	5.4	2.7	$\underline{2.5}$	2.2	3.3
	w=4	4.1	3.1	$\underline{1.8}$	1.6	1.5	1.9	w=4	1.8	2.1	$\underline{1.6}$	1.5	1.5	1.5
	w=8	1.7	$\underline{1.4}$	1.0	1.2	1.0	1.0	w=8	1.0	$\underline{1.2}$	1.0	1.2	1.2	1.0
	$AvgEI_l$	4.7	2.7	1.3	1.2	1.0		$AvgTI_l$	1.9	1.7	1.1	1.1	1.0	
3SI	w=1	162.3	12.8	7.1	5.4	$\underline{6.0}$	22.8	w=1	51.5	5.8	4.7	4.1	$\underline{4.4}$	12.0
	w=2	17.7	5.1	2.8	$\underline{2.7}$	3.2	4.4	w=2	6.4	2.7	2.0	$\underline{2.0}$	2.6	2.8
	w=4	4.1	2.4	$\underline{1.5}$	1.4	1.6	1.7	w=4	1.9	1.7	$\underline{1.3}$	1.3	1.5	1.4
	w=8	1.9	$\underline{1.2}$	1.1	1.0	1.0	1.0	w=8	1.2	$\underline{1.1}$	1.2	1.0	1.1	1.0
	$AvgEI_l$	9.3	1.6	1.0	0.9	1.0		$AvgTI_l$	4.1	1.1	0.9	0.9	1.0	
2BX	w=1	93.8	17.2	7.8	6.0	$\underline{5.4}$	12.8	w=1	31.5	7.8	4.1	3.5	$\underline{3.8}$	6.7
	w=2	12.3	6.0	3.2	$\underline{2.7}$	2.6	3.3	w=2	5.7	3.5	2.1	$\underline{1.9}$	2.3	2.3
	w=4	6.4	2.9	$\underline{2.1}$	1.5	1.9	1.9	w=4	3.5	1.9	$\underline{1.5}$	1.2	1.9	1.5
	w=8	2.7	$\underline{1.5}$	1.2	1.0	1.0	1.0	w=8	1.9	$\underline{1.2}$	1.0	1.0	1.3	1.0
	$AvgEI_l$	7.0	2.1	1.2	1.0	1.0		$AvgTI_l$	3.5	1.4	0.9	0.8	1.0	
3EP	w=1	1802.5	155.0	71.2	69.3	$\underline{79.9}$	200.2	w=1	478.6	54.1	33.4	37.6	$\underline{48.2}$	100.7
	w=2	35.5	12.1	8.8	$\underline{7.6}$	8.6	8.9	w=2	12.3	5.7	5.1	$\underline{5.0}$	6.8	6.0
	w=4	8.8	3.8	$\underline{2.4}$	2.6	2.6	2.6	w=4	3.8	2.1	$\underline{1.6}$	2.0	2.4	2.0
	w=8	2.6	$\underline{1.4}$	1.2	1.2	1.0	1.0	w=8	1.4	$\underline{1.0}$	1.0	1.1	1.2	1.0
	$AvgEI_l$	8.2	1.6	1.0	1.0	1.0		$AvgTI_l$	3.6	0.9	0.7	0.8	1.0	
4EP	w=1	>649.8	649.8	117.2	71.9	$\underline{78.5}$	>173.2	w=1	>200.0	200.0	50.4	34.3	$\underline{44.2}$	>82.7
	w=2	21.9	7.6	4.2	$\underline{4.3}$	5.6	5.0	w=2	5.9	3.4	2.4	$\underline{2.8}$	4.2	3.0
	w=4	3.2	1.9	$\underline{1.5}$	1.6	1.5	1.3	w=4	1.1	1.1	$\underline{1.0}$	1.2	1.2	0.9
	w=8	2.8	$\underline{1.6}$	1.4	1.1	1.0	1.0	w=8	1.5	$\underline{1.2}$	1.3	1.0	1.1	1.0
	$AvgEI_l$	>4.3	3.1	1.2	1.0	1.0		$AvgTI_l$	>2.1	1.8	0.9	0.8	1.0	
5EP	w=1	>551	>551	551	>551	$\underline{>551}$	>428.7	w=1	>408	>408	408	>408	$\underline{>408}$	>213.1
	w=2	>551	5.8	2.8	$\underline{3.5}$	4.7	>36.4	w=2	>408	3.8	2.8	$\underline{4.2}$	5.6	>31.0
	w=4	16.5	1.1	$\underline{1.0}$	1.0	1.1	1.8	w=4	9.0	1.0	$\underline{1.4}$	1.5	1.6	1.3
	w=8	3.3	$\underline{1.3}$	1.1	1.1	1.0	1.0	w=8	2.8	$\underline{1.7}$	1.8	1.8	1.8	1.0
	$AvgEI_l$	>34.1	>1.1	0.9	>0.9	>1.0		$AvgTI_l$	>20.3	>0.8	0.8	>0.9	>1.0	
5SM	w=1	>23.4	20.0	11.2	11.1	$\underline{12.5}$	>9.5	w=1	>7.7	7.6	6.0	7.2	$\underline{7.8}$	>5.5
	w=2	23.4	8.0	5.6	$\underline{5.0}$	3.4	4.6	w=2	7.7	3.9	3.8	$\underline{3.7}$	2.4	3.0
	w=4	8.0	3.1	$\underline{3.1}$	2.2	2.3	2.1	w=4	3.3	2.0	$\underline{2.8}$	2.0	2.1	1.8
	w=8	3.5	$\underline{2.2}$	1.2	1.0	1.1	1.0	w=8	1.9	$\underline{1.9}$	1.2	1.0	1.1	1.0
	$AvgEI_l$	>3.9	1.8	1.2	1.1	1.0		$AvgTI_l$	>1.9	1.3	1.2	1.1	1.0	
FIB	w=1	137.2	47.7	48.4	17.1	$\underline{13.1}$	46.3	w=1	88.9	35.3	35.8	13.4	$\underline{11.6}$	26.2
	w=2	7.3	10.9	5.1	$\underline{3.6}$	2.5	4.9	w=2	4.6	10.0	5.1	$\underline{3.8}$	3.0	3.0
	w=4	3.9	4.9	$\underline{2.4}$	2.0	2.1	2.6	w=4	3.4	5.6	$\underline{3.0}$	2.6	2.9	2.0
	w=8	1.0	$\underline{1.2}$	1.7	1.3	1.0	1.0	w=8	1.0	$\underline{1.9}$	3.1	2.4	1.8	1.0
	$AvgEI_l$	4.1	2.9	2.1	1.2	1.0		$AvgTI_l$	2.7	2.3	1.9	1.2	1.0	
EPT	w=1	66.7	12.7	7.2	5.9	$\underline{5.6}$	15.2	w=1	21.4	6.4	5.6	5.1	$\underline{5.0}$	6.2
	w=2	11.1	3.5	2.2	$\underline{1.7}$	1.4	3.1	w=2	4.7	2.6	2.0	$\underline{1.9}$	1.5	1.6
	w=4	4.2	1.8	$\underline{1.3}$	2.0	1.0	1.6	w=4	3.1	1.5	$\underline{1.6}$	2.6	1.4	1.2
	w=8	1.3	$\underline{1.0}$	1.7	1.4	1.6	1.0	w=8	1.0	$\underline{1.5}$	3.3	2.5	3.4	1.0
	$AvgEI_l$	6.2	1.8	1.3	1.3	1.0		$AvgTI_l$	2.5	1.1	1.1	1.2	1.0	

Table 10: $RelE$ and $RelT$ values of the 14 benchmark problems. The extreme left column shows the abbreviations of the 14 problems. The underlined figures represent MAPs with at most 128 sub-instructions ($w \times l = 128$).

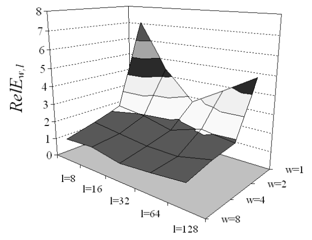
	Relative Computational Effort ($RelE$)							Relative Evolution Time ($RelT$)						
		l=8	l=16	l=32	l=64	l=128	$AvgEA_w$		l=8	l=16	l=32	l=64	l=128	$AvgTA_w$
ANT	w=1	29.0	9.3	7.4	3.6	$\underline{4.0}$	*8.3*	w=1	8.6	4.3	4.4	2.8	$\underline{3.6}$	*3.9*
	w=2	6.1	2.2	1.5	$\underline{1.2}$	1.1	*1.9*	w=2	3.1	1.3	1.1	$\underline{1.1}$	1.1	*1.3*
	w=4	2.6	1.5	$\underline{1.2}$	1.2	1.1	*1.2*	w=4	1.5	1.5	$\underline{1.2}$	1.1	1.2	*1.1*
	w=8	1.3	$\underline{1.4}$	1.4	1.1	1.0	*1.0*	w=8	1.1	$\underline{1.0}$	1.7	1.2	1.4	*1.0*
	$AvgEI_l$	*4.1*	*1.8*	*1.4*	*1.0*	*1.0*		$AvgTI_l$	*1.8*	*1.1*	*1.1*	*0.9*	*1.0*	
BCW	w=1	4.6	3.1	3.6	5.3	$\underline{8.5}$	*4.2*	w=1	1.5	1.4	2.0	3.3	$\underline{5.2}$	*1.2*
	w=2	2.9	2.9	3.3	$\underline{2.9}$	3.2	*2.5*	w=2	1.0	1.4	2.0	$\underline{1.9}$	2.2	*0.7*
	w=4	2.4	2.4	$\underline{2.0}$	2.2	2.0	*1.8*	w=4	1.0	1.5	$\underline{1.6}$	1.9	1.8	*0.7*
	w=8	1.7	$\underline{1.2}$	1.3	1.0	1.1	*1.0*	w=8	1.8	$\underline{2.1}$	2.6	2.3	2.5	*1.0*
	$AvgEI_l$	*1.0*	*0.9*	*0.9*	*0.9*	*1.0*		$AvgTI_l$	*0.5*	*0.6*	*0.8*	*0.9*	*1.0*	
BLD	w=1	5.8	4.8	6.0	6.6	$\underline{7.2}$	*5.0*	w=1	2.9	3.3	4.8	5.9	$\underline{7.0}$	*1.7*
	w=2	2.8	2.6	2.0	$\underline{3.3}$	3.7	*2.3*	w=2	1.5	1.8	1.6	$\underline{3.0}$	3.7	*0.8*
	w=4	1.5	1.6	$\underline{1.8}$	2.4	2.1	*1.5*	w=4	1.0	1.4	$\underline{1.9}$	2.7	2.6	*0.7*
	w=8	1.5	$\underline{1.4}$	1.0	1.3	1.1	*1.0*	w=8	2.1	$\underline{2.7}$	2.4	3.5	3.0	*1.0*
	$AvgEI_l$	*0.9*	*0.9*	*0.8*	*1.0*	*1.0*		$AvgTI_l$	*0.5*	*0.6*	*0.7*	*1.0*	*1.0*	

Table 10 (continue): $RelE$ and $RelT$ values of the 14 benchmark problems.

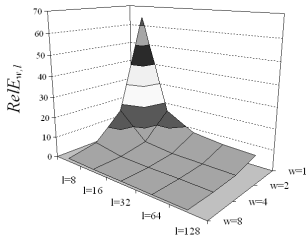
In Table 10, the experimental results of the four single-input numeric functions (including CUB, SEX, 3PI, and 3SI) show that both $AvgEA_w$ and $AvgTA_w$ of these experiments monotonically decrease while w increases from 1 to 8. For example, $AvgEA_w$ of CUB monotonically decreases from 3.8 to 1.0 while w increases from 1 to 8. In other words, the more ALUs used, the less the computational effort (E) . The value 3.8 means that the computational effort to evolve a 1-ALU sequential program is, on average, 3.8 times that of an 8-ALU parallel program. Similarly, $AvgTA_w$ of CUB monotonically decreases from 2.7 to 1.0 while w increases from 1 to 8. Thus, the required evolution time (T) of a 1-ALU sequential program is, on average, 2.7 times that of an 8-ALU parallel program. The $AvgEA_1$ values of SEX, 3PI and 3SI are 15.6, 17.9, and 22.8 respectively. The $AvgTA_1$ values of SEX, 3PI and 3SI are 8.7, 10.5 and 12.0 respectively. We conclude that GPP reduces E and T super-linearly (>8 times) with respect to the number of ALUs of the three problems. For the Two-box (2BX) function, both $AvgEA_w$ and $AvgTA_w$ monotonically decrease while w increases from 1 to 8. The values of $AvgEA_1$ and $AvgTA_1$ are 12.8 and 6.7 respectively.

From the experimental results of the four Boolean functions (including 3EP, 4EP, 5EP, and 5SM) shown in Table 10, both $AvgEA_w$ and $AvgTA_w$ monotonically decrease while w increases from 1 to 8. Noticeably, GPP did not evolve any satisfactory solution program for the 5EP problems in the low resources region (with small w and/or l values). This is because that GPP had to construct the *xor* function from the four primitive functions (*and*, *or*, *nand* and *nor*) used in the ALU function sets (F). Therefore, a solution program had to be long and complicated. By comparing the results with the 3EP and 4EP experiments, we can see that the difficulties of even-parity problems increase when the number of inputs increases. The actual values of $AvgEA_1$ and $AvgTA_1$ of 5EP should be much greater than the values obtained in the 3EP and 4EP experiments. The $AvgEA_1$ values of 3EP, 4EP, 5EP, and 5SM are 200.2, >173.2, >428.7, and >9.5 respectively. The $AvgTA_1$ values of 3EP, 4EP, 5EP, and 5SM are 100.7, >82.7, >213.1, and >5.5 respectively. We conclude that GPP reduces super-linearly both E and T of these four problems.

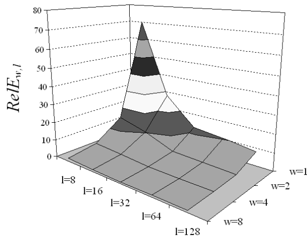
Detailed experimental results of the remaining five experiments (including FIB, EPT, ANT, BCW, and BLD) are shown in the corresponding rows in Table 10. Similar to the previously presented results, both $AvgEA_w$ and $AvgTA_w$ monotonically decrease while w increases from 1 to 8.



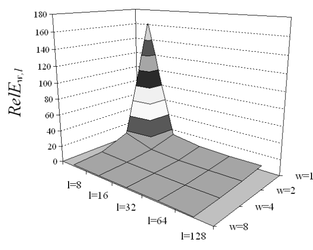
(a) *Cubic* (CUB)



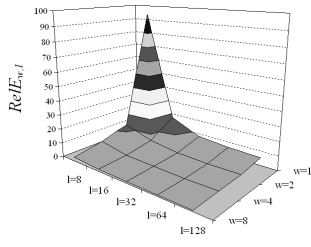
(b) *Sextic* (SEX)



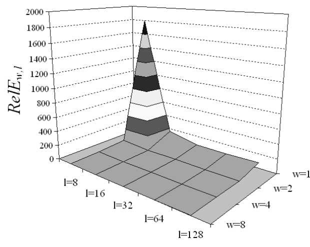
(c) *3-*π (3PI)



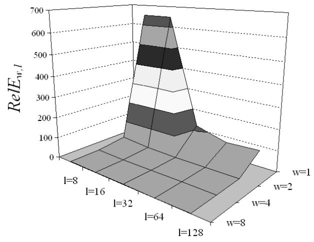
(d) *3-sine* (3SI)



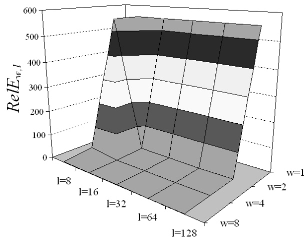
(e) *Two-box* (2BX)



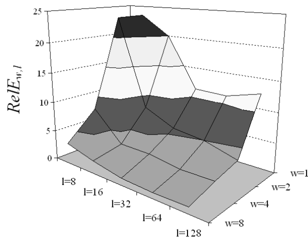
(f) *3-even-parity* (3EP)



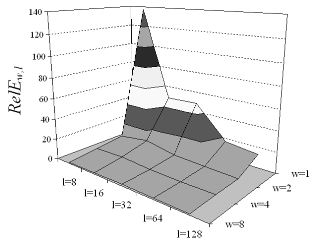
(g) *4-even-parity** (4EP)



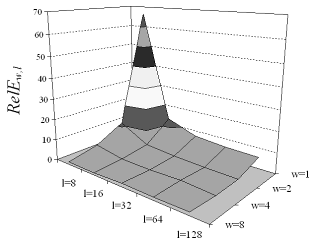
(h) *5-even-parity** (5EP)



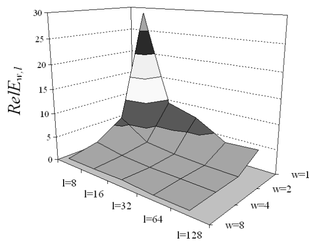
(i) *5-symmetry** (5SM)



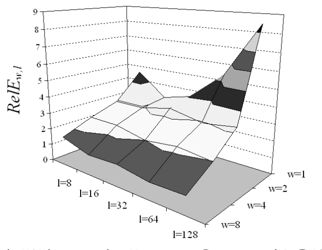
(j) *Finonacci* (FIB)



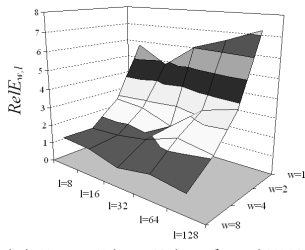
(k) *32-bit Even-Parity* (EPT)



(l) *Artificial Ant* (ANT)

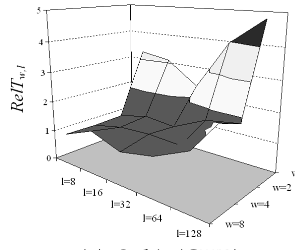


(m) *Wisconsin Breast Cancer* (BCW)

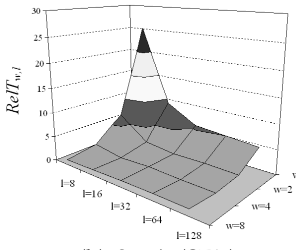


(n) *Bupa Liver Disorders* (BLD)

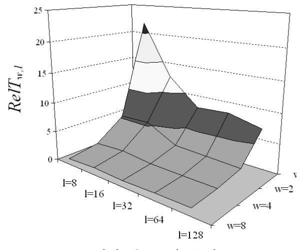
Figure 7: $RelE_{w,l}$ of the 14 problems versus w and l. Items marked with * have at least one MAP setting which did not evolve any satisfactory program in all 100 runs.



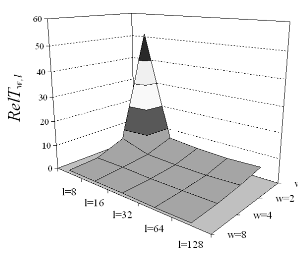
(a) *Cubic* (CUB)



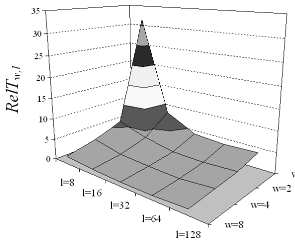
(b) *Sextic* (SEX)



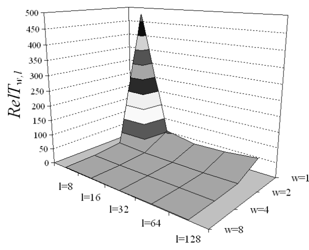
(c) *3-π* (3PI)



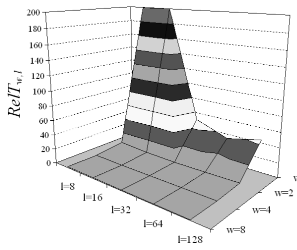
(d) *3-sine* (3SI)



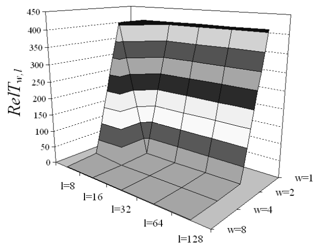
(e) *Two-box* (2BX)



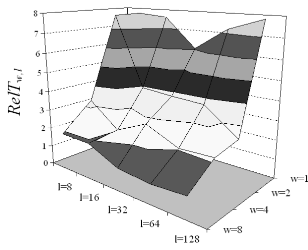
(f) *3-even-parity* (3EP)



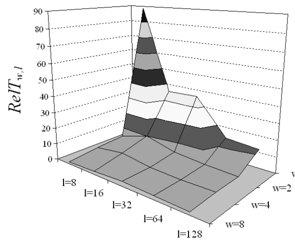
(g) *4-even-parity** (4EP)



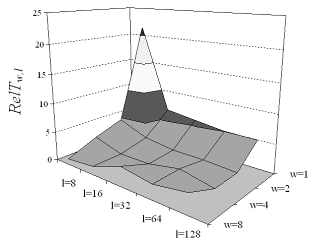
(h) *5-even-parity** (5EP)



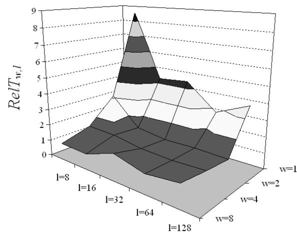
(i) *5-symmetry** (5SM)



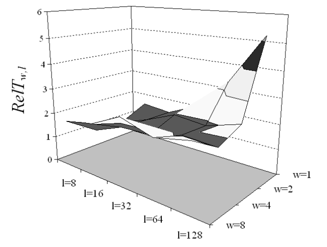
(j) *Finonacci* (FIB)



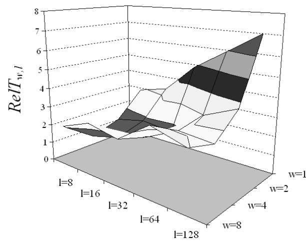
(k) *32-bit Even-Parity* (EPT)



(l) *Artificial Ant* (ANT)



(m) *Wisconsin Breast Cancer* (BCW)

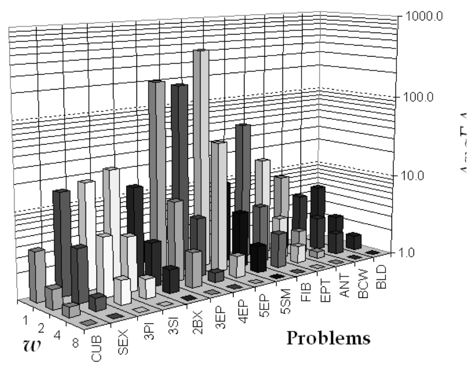


(n) *Bupa Liver Disorders* (BLD)

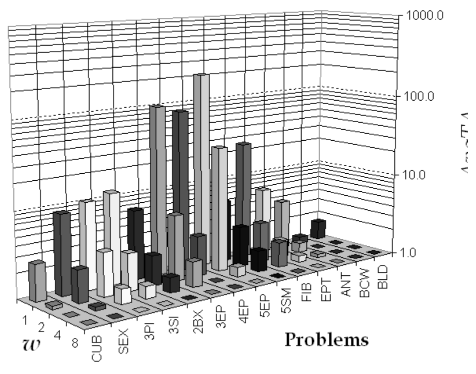
Figure 8: $RelT_{w,l}$ of the 14 problems versus w and l. Items marked with * have at least one MAP setting which did not evolve any satisfactory program in all 100 runs.

	$AvgEA_w$					$AvgTA_w$			
	w=1	w=2	w=4	w=8		w=1	w=2	w=4	w=8
CUB	3.8	1.7	1.3	1.0	CUB	2.7	1.1	1.0	1.0
SEX	15.6	4.2	1.4	1.0	SEX	8.7	2.4	1.1	1.0
3PI	17.9	5.0	1.9	1.0	3PI	10.5	3.3	1.5	1.0
3SI	22.8	4.4	1.7	1.0	3SI	12.0	2.8	1.4	1.0
2BX	12.8	3.3	1.9	1.0	2BX	6.7	2.3	1.5	1.0
3EP	200.2	8.9	2.6	1.0	3EP	100.7	6.0	2.0	1.0
4EP	>173.2	5.0	1.3	1.0	4EP	>82.7	3.0	0.9	1.0
5EP	>428.7	>36.4	1.8	1.0	5EP	>213.1	>31.0	1.3	1.0
5SM	>9.5	4.6	2.1	1.0	5SM	>5.5	3.0	1.8	1.0
FIB	46.3	4.9	2.6	1.0	FIB	26.2	3.0	2.0	1.0
EPT	15.2	3.1	1.6	1.0	EPT	6.2	1.6	1.2	1.0
ANT	8.3	1.9	1.2	1.0	ANT	3.9	1.3	1.1	1.0
BCW	4.2	2.5	1.8	1.0	BCW	1.2	0.7	0.7	1.0
BLD	5.0	2.3	1.5	1.0	BLD	1.7	0.8	0.7	1.0
AVGE$_w$	*68.8*	*6.3*	*1.8*	*1.0*	AVGT$_w$	*34.4*	*4.5*	*1.3*	*1.0*
$GainEA_w$	*1.0*	*10.9*	*38.2*	*68.8*	$GainTA_w$	*1.0*	*7.6*	*26.5*	*34.4*

Table 11: $AvgEA_w$ and $AvgTA_w$ of the 14 problems versus w. AVGE$_w$ and AVGT$_w$ are the averages of $AvgEA_w$ and $AvgTA_w$ values respectively over the 14 benchmark problems. $GainEA_w$=AVGE$_1$/AVGE$_w$ and $GainTA_w$=AVGT$_1$/AVGT$_w$.



(a)



(b)

Figure 9: $AvgEA_w$ and $AvgTA_w$ versus w. Values are presented in logarithmic scale.

Based on the experimental results discussed in this section, it is observed that there is an evident trend of decreasing E and T with an increase of w. A MAP with larger w does reduce computational effort (E) and evolution time (T). We shall discuss the integrated summary results in the following subsections.

6.3 E and T versus w

This section presents the variations of the two performance indicators (E and T) versus different numbers of ALUs (w : width). We copy all $AvgEA_w$ and $AvgTA_w$ values from Table 10 to Table 11. The graphical representations are given in Figure 9. Undoubtedly, all $AvgEA_w$ values show an evident trend of decreasing with an increasing w. The gain of E ($GainEA_w$) of a w-ALU program compared to that of a 1-ALU sequential program is shown in the $GainEA_w$ row of the table. The $GainEA_w$ of 2-, 4-, and 8-ALU parallel programs are 10.9, 38.2, and 68.8 times, respectively, of a 1-ALU sequential program. According to these values, the $GainEA_w$ increases super-linearly with respect to w. Thus, it is worth it to spend more hardware resources to build more ALUs in order to reduce the computational effort.

Similarly, $AvgTA_w$ values listed in Table 11 show an evident trend of decreasing T with an increasing w. One concern related to GPP is that the complicated steps in evaluating a parallel program will cancel out the computational effort gained. However, as shown in the table, the gain of T ($GainTA_w$) of 2-, 4-, and 8-ALU parallel programs are 7.6, 26.5, and 34.4 times, respectively, of a 1-ALU sequential program. It demonstrates the effectiveness and efficiency of the GPP paradigm with a parallel program representation. It also shows that GPP speeds up evolution not only by hardware-assisted fitness evaluations but also by its robust parallel representation form. It opens up a new opportunity to use GPP to solve more complicated problems by including higher level functions, such as problem specific modules. Even though these higher level functions may not be implemented in hardware, the pure software speedup of GPP still benefits the evolution.

6.4 E and T versus l

This section presents the variations of E and T versus different numbers of parallel-instructions (l : length) allowed in the genetic programs. Table 12 and Figure 10 show all the $AvgEI_l$ and $AvgTI_l$ values copied from Table 10. The gain of E ($GainEI_l$) and gain of T ($GainTI_l$) of an l-parallel-instruction program compared to that of an 8-parallel-instruction program are shown in the $GainEI_l$ and $GainTI_l$ rows respectively. The $GainEI_l$ of 16-, 32-, 64-, and 128-parallel-instruction programs are 3.8, 5.8, 6.9, and 6.9 times, respectively, of an 8-parallel-instruction program. According to these values, it shows a general trend of a decreasing E with an increasing l. However, the improvement of E versus l is not as high as that versus w (see Table 11). It shows that the trend is also saturating. The $GainTI_l$ of 16-, 32-, 64-, and 128-parallel-instruction programs are 2.9, 3.5, 3.5, and 3.5 times, respectively, of an 8-parallel-instruction program. According to these values, increasing l only increases T slightly and it shows that the trend is also saturating.

6.5 E and T versus Different Combinations of w and l

This section presents the variations of E and T versus different combinations of w and l. We compare four different MAPs with a maximum of 128 ALU sub-instructions (i.e. $w \times l = 1 \times 128$, 2×64, 4×32, and 8×16). These MAPs have similar sizes of search space (genotype bit-size). In order to compare the results of the 14 problems, we divide all $RelE_{w,l}$ and $RelT_{w,l}$ values, which are underlined in the diagonal cells in Table 10, by $RelE_{8,16}$ and $RelT_{8,16}$ respectively (see Table 13 and Figure 11). Undoubtedly, all relative computational efforts ($RelE$) and relative evolution time ($RelT$) values show evident trends of decreasing E and T with respect to increasing w. AVGE$_{w,l}$ decreases from 42.1 to 1.0 and AVGT$_{w,l}$ decreases from 26.2 to 1.0 while w increases from 1 to 8. It shows that a wide-and-short program is easier to evolve than a long-and-narrow program. It demonstrates the effectiveness and efficiency of the GPP paradigm with a parallel program representation.

	AvgEI$_l$				
	l=8	l=16	l=32	l=64	l=128
CUB	1.2	1.0	0.7	0.9	1.0
SEX	7.0	2.0	1.2	1.1	1.0
3PI	4.7	2.7	1.3	1.2	1.0
3SI	9.3	1.6	1.0	0.9	1.0
2BX	7.0	2.1	1.2	1.0	1.0
3EP	8.2	1.6	1.0	1.0	1.0
4EP	>4.3	3.1	1.2	1.0	1.0
5EP	>34.1	>1.1	0.9	>0.9	>1.0
5SM	>3.9	1.8	1.2	1.1	1.0
FIB	4.1	2.9	2.1	1.2	1.0
EPT	6.2	1.8	1.3	1.3	1.0
ANT	4.1	1.8	1.4	1.0	1.0
BCW	1.0	0.9	0.9	0.9	1.0
BLD	0.9	0.9	0.8	1.0	1.0
AVGE$_l$	*6.9*	*1.8*	*1.2*	*1.0*	*1.0*
GainEI$_l$	*1.0*	*3.8*	*5.8*	*6.9*	*6.9*

	AvgTI$_l$				
	l=8	l=16	l=32	l=64	l=128
CUB	0.6	0.7	0.6	0.8	1.0
SEX	3.2	1.4	1.1	1.1	1.0
3PI	1.9	1.7	1.1	1.1	1.0
3SI	4.1	1.1	0.9	0.9	1.0
2BX	3.5	1.4	0.9	0.8	1.0
3EP	3.6	0.9	0.7	0.8	1.0
4EP	>2.1	1.8	0.9	0.8	1.0
5EP	>20.3	>0.8	0.8	>0.9	>1.0
5SM	>1.9	1.3	1.2	1.1	1.0
FIB	2.7	2.3	1.9	1.2	1.0
EPT	2.5	1.1	1.1	1.2	1.0
ANT	1.8	1.1	1.1	0.9	1.0
BCW	0.5	0.6	0.8	0.9	1.0
BLD	0.5	0.6	0.7	1.0	1.0
AVGT$_l$	*3.5*	*1.2*	*1.0*	*1.0*	*1.0*
GainTI$_l$	*1.0*	*2.9*	*3.5*	*3.5*	*3.5*

Table 12: *AvgEI*$_l$ and *AvgTI*$_l$ of the 14 problems versus l. AVGE$_l$ and AVGT$_l$ are the averages of *AvgEI*$_l$ and *AvgTI*$_l$ values respectively over the 14 benchmark problems. *GainEI*$_l$=AVGE$_8$/AVGE$_l$ and *GainTI*$_l$=AVGT$_8$/AVGT$_l$.

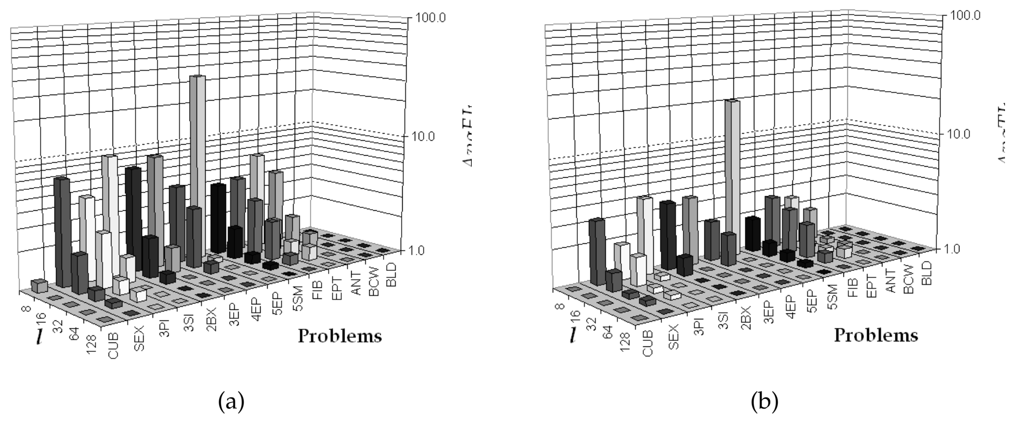


Figure 10: *AvgEI*$_l$ and *AvgTI*$_l$ versus l. Values are presented in logarithmic scale.

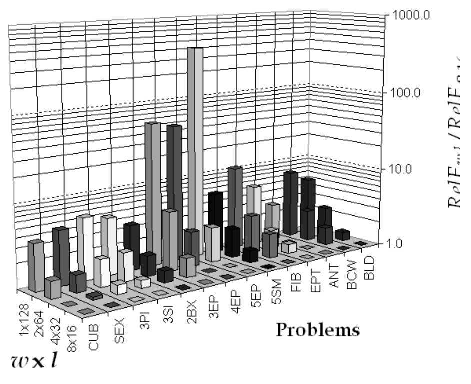
7 Summary and Discussions

An extensive study on the effectiveness and efficiency of the GPP paradigm has been performed on 14 benchmark problems. An in-depth investigation of GPP based on two performance indicators, the Kozas computational effort (E) and the pure software emulator execution time (T), have been presented. Experimental results show that both E and T decrease if the program width (the number of ALUs) increases. For example, evolving a 1-ALU sequential program requires, on average, 68.8 times the computational effort of an 8-ALU parallel program (see the last row in Table 11). It indicates that it is easier to evolve a parallel program than its sequential counterpart. In addition, experimental results show that evolving a 1-ALU sequential program requires, on average, 34.4 times in the pure software emulator execution time of an 8-ALU parallel program (see the last row in Table 11). It demonstrates the effectiveness and efficiency of the GPP paradigm with a parallel program representation. The evolutionary speedup is not only given by hardware-assisted fitness evaluations but also by its internal robust parallel representation form. This GPP paradigm can be extended to solve more complicated problems by including higher level functions which could be application specific. Not all these higher level functions need to be implemented in hardware. The pure software approach can still benefit from the accelerating property of the parallel representation.

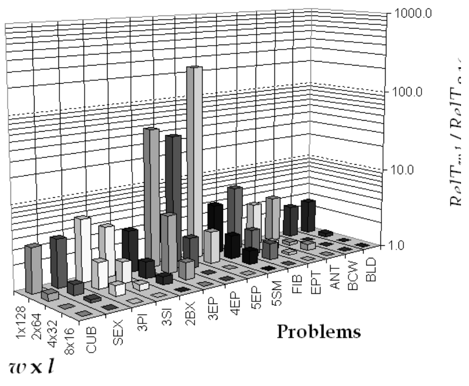
	$RelE_{w,l}/RelE_{8,16}$			
	1×128	2×64	4×32	8×16
CUB	3.7	1.6	1.0	1.0
SEX	4.6	1.6	1.1	1.0
3PI	5.7	2.1	1.3	1.0
3SI	5.0	2.3	1.3	1.0
2BX	3.6	1.8	1.4	1.0
3EP	57.1	5.4	1.7	1.0
4EP	49.1	2.7	0.9	1.0
5EP	>423.8	2.7	0.8	1.0
5SM	5.7	2.3	1.4	1.0
FIB	10.9	3.0	2.0	1.0
EPT	5.6	1.7	1.3	1.0
ANT	2.9	0.9	0.9	1.0
BCW	7.1	2.4	1.7	1.0
BLD	5.1	2.4	1.3	1.0
AVGE$_{w,l}$	*42.1*	*2.3*	*1.3*	*1.0*

	$RelT_{w,l}/RelT_{8,16}$			
	1×128	2×64	4×32	8×16
CUB	3.4	1.2	0.9	1.0
SEX	3.8	1.3	1.1	1.0
3PI	5.7	2.1	1.3	1.0
3SI	4.0	1.8	1.2	1.0
2BX	3.2	1.6	1.3	1.0
3EP	48.2	5.0	1.6	1.0
4EP	36.8	2.3	0.8	1.0
5EP	>240.0	2.5	0.8	1.0
5SM	4.1	1.9	1.5	1.0
FIB	6.1	2.0	1.6	1.0
EPT	3.3	1.3	1.1	1.0
ANT	3.6	1.1	1.2	1.0
BCW	2.5	0.9	0.8	1.0
BLD	2.6	1.1	0.7	1.0
AVGT$_{w,l}$	*26.2*	*1.9*	*1.1*	*1.0*

Table 13: $RelE_{w,l}/RelE_{8,16}$ and $RelT_{w,l}/RelT_{8,16}$ values of MAPs with $w \times l$=128. AVGE$_{w,l}$ and AVGT$_{w,l}$ show the average values over the 14 problems.



(a)



(b)

Figure 11: $RelE_{w,l}/RelE_{8,16}$ and $RelT_{w,l}/RelT_{8,16}$ versus different combinations of $w \times l$. Values are presented in logarithmic scale.

8 Conclusions

This paper has presented a novel Genetic Parallel Programming (GPP) paradigm based on linear GP that can be used to evolve optimal parallel programs based on parallel architecture. It has also presented the evolutionary speedup on the GPP paradigm – parallel programs can be evolved more efficiently than sequential programs. The phenomenon shows that GPP does not only evolve parallel programs automatically, but it also boosts the pure software evolution speed. It also demonstrates an increase in the efficiency of linear GP and leads to a new approach in evolving a parallel program with an MIMD processor, such as the Multi-ALU Processor (MAP). This will have significant impact on GP evolution research. Furthermore, it opens up an entirely new approach to evolving a satisfactory solution in parallel program representation and then serializing it into sequential codes. The serialization is a one-time process and its computational effort is linearly proportional to the size of the solution parallel program. As a result, the whole evolution process of standard GP can be sped up.

Acknowledgement

The work described in this paper was partially supported by two grants from the Research Grants Council of Hong Kong Special Administrative Region, China (Project No. CUHK4192/03E and CUHK4127/04E)

References

Andre, D. and Koza, J. R. (1996). Parallel genetic programming: a scalable implementation using the transputer network architecture. In Angeline, P. J. et al., editors, *Advances in Genetic Programming 2*, pages 317–337, MIT Press.

Andre, D. and Koza, J. R. (1997). Exploiting the fruits of parallelism: an implementation of parallel genetic programming that achieves super-linear performance. *Information Science Journal*, 106(3-4):201–218.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming: An Introduction on the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann.

Banzhaf, W., Koza, J. R., Ryan, C., Spector, L., and Jocob, C. (2000). Genetic programming. *IEEE Intelligent Systems Journal*, 15(3):74–84.

Brameier, M. and Banzhaf, W. (2001). A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26.

Brameier, M., Hoffmann, F., Nordin, P., Banzhaf, W., and Francone, F. (1999). Parallel machine code genetic programming. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of 1999 Genetic and Evolutionary Computation Conference*, pages 1228-1228, Morgan Kaufmann.

Cheang, S. M. (2003). An empirical study of the gpp accelerating phenomenon. In Vadakkepat, P., Wan, T. W., Chen, T. K., and Poh, L. A., editors, *Proceedings of the Second International Conference on Computational Intelligence, Robotics and Autonomous Systems*, pages PS04-4-03, National University of Singapore.

Cheang, S. M., Lee, K. H., and Leung, K. S. (2003). Evolving data classification programs using genetic parallel programming. In *Proceedings of IEEE 2003 Congress on Evolutionary Computation*, pages 248–255, IEEE Press.

Cheang, S. M., Lee, K. H., and Leung, K. S. (2004). Designing optimal combinational digital circuits using a multiple logic unit processor. In Keijzer, M., OReilly, U., Lucas, S. M., Costa, E., and Soule, T., editors, *Proceedings of the Seventh European Conference on Genetic Programming*, pages 23–34, Springer-Verlag.

Conrads, M., Nordin, P., and Banzhaf, W. (1998). Speech sound discrimination with genetic programming. In Banzhaf, W., Poli, R., Schoenauer, M., and Fogarty, T. C., editors, *Proceedings of the First European Workshop on Genetic Programming*, pages 113–129, Springer-Verlag.

Folino, G., Pizzuti, C., and Spezzano, G. (2003). A scalable cellular implementation of parallel genetic programming. *IEEE Transactions on Evolutionary Computation*, 7(1):37–53.

Francone, F. D. (2001). *Discipulus^{TM} 3.0 Owners Manual*, Register Machine Learning Technologies, Inc. [http://www.aimlearning.com]

Heywood, M. I. and Zincir-Heywood, A. N. (2000). Register based genetic programming on fpga computing platforms. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P., and Fogarty, T. C., editors, *Proceedings of the Third European Conference on Genetic Programming*, pages 44–58, Springer-Verlag.

Heywood, M. I. and Zincir-Heywood, A. N. (2000a). Page-based linear genetic programming. In *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics*, pages 3823–3828. IEEE Press.

Heywood, M. I. and Zincir-Heywood, A. N. (2002). Dynamic page based crossover in linear genetic programming. *IEEE Transactions on Systems, Man, and Cybernetics - Part B*, 32(3):380–388.

Huelsbergen, L. (1997). Learning recursive sequences via evolution of machine-language programs. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Proceedings of the Second Annual Conference on Genetic Programming*, pages 186–194, Morgan Kaufmann.

Juille, H. and Pollack, J. B. (1995). Parallel genetic programming and fine-grained SIMD architecture. In Siegel, E. V. et al., editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 31–37, MIT Press.

Kalganova, T. and Miller, J. F. (1999). Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pages 54–63, IEEE Press.

Kantschik, W. and Banzhaf, W. (2001). Linear-tree gp and its comparison with other gp structures. In Miller, J., Tomassini, M., Lanzi, P. L., Ryan, C., Tettamanzi, A. G. B., and Langdon, W. B., editors, *Proceedings of the Fourth European Conference on Genetic Programming*, pages 303–312, Springer-Verlag.

Kantschik, W. and Banzhaf, W. (2002). Linear-graph gp – a new gp structure. In Foster, J. A., Lutton, E., Miller, J., Ryan, C., and Tettamanzi, A. G. B., editors, *Proceedings of the Fifth European Conference on Genetic Programming*, pages 83–92, Springer-Verlag.

Kishore, J. K., Patnaik, L. M., Mani, V., and Agrawal, V. K. (2000). Application of genetic programming for multicategory pattern classification. *IEEE Transactions on Evolutionary Computation*, 4(3):242–258.

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

Koza, J. R., Bennett III, F. H., Andre, D., and Keane, M. A. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann.

Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., and Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic.

Krawiec, K. and Bhanu, B. (2003). Coevolution and linear genetic programming for visual learning. In Cantú-Paz, E. et al., editors, *Proceedings of 2003 Genetic and Evolutionary Computation Conference*, pages 332–343, Springer-Verlag.

Lau, W. S., Li, G., Lee, K. H., Leung, K. S. and Cheang, S. M. (2005). Multi-logic-unit processor : a combinational logic circuit evaluation engine for genetic parallel programming. In Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., and Tomazzini, M., editors, *Proceedings of the eighth European Conference on Genetic Programming*, pages 167–177, Springer-Verlag.

Lee, K. H., Leung, K. S., and Cheang, S. M. (1990). A Microprogrammable List Processor for Personal Computers. *IEEE Micro Journal*, 10(4):50–61.

Leung, K. S., Lee, K. H., and Cheang, S. M. (2002). Genetic parallel programming – evolving linear machine codes on a multiple-ALU processor. In Yaacob, S., Nagarajan, M., and Chekima, A., editors, *Proceedings of the International Conference on Artificial Intelligence in Engineering and Technology*, pages 207–213, University of Malaysia Sabah.

Leung, K. S., Lee, K. H., and Cheang, S. M. (2002a). Evolving parallel machine programs for a multi-ALU processor. In *Proceedings of 2002 Congress on Evolutionary Computation*, pages 1703–1708, IEEE Press.

Leung, K. S., Lee, K. H., and Cheang, S. M. (2003). Parallel programs are more evolvable than sequential programs. In Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., and Costa E., editors, *Proceedings of the Sixth European Conference on Genetic Programming*, pages 107–118, Springer-Verlag.

Lim, T. S., Loh, W. Y., and Shih, Y. S. (2000). A Comparison of Prediction Accuracy, Complexity, and Training Time of Thirty-Three Old and New Classification Algorithms. *Machine Learning*, Kluwer Academic, 40:203–229.

Mahfoud, S. W. (1992). Crowding and preselection revisited. In Manner, R. and Manderick, B., editors, *Proceedings of Conference on Parallel Problem Solving from Nature*, pages 27–36, Elsevier Science Publishers.

Martin, W. N., Lienig, J., and Cohoon, J. P. (1997). Island (migration) models: evolutionary algorithms based in punctuated equilibria. In Bäck, T. et al., editors, *Handbook of Evolutionary Computation*, pages C6.3:1–16, Oxford University Press.

Merz, C. J. and Murphy, P. M. (1996). *UCI Repository of Machine Learning Databases*, 1996. University of California.

Miller, J. F. (1999). An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of 1999 Genetic and Evolutionary Computation Conference*, pages 1135–1142, Morgan Kaufmann.

Miller, J. F. and Thomson, P. (2000). Cartesian genetic programming. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P., and Fogarty, T. C., editors, *Proceedings of the Third European Conference on Genetic Programming*, pages 121–132, Springer-Verlag.

Miller, J. F., Job, D., and Vassilev, V. K. (2000). Principles in the evolutionary design of digital circuits – part I. *Genetic Programming and Evolvable Machines*, 1(1):7–35.

Moore, G. E. (1996). Can Moore's Law Continue Indefinitely? *Computerworld Leadership Series*, 2(6):2–7.

Muni, D. P., Pal, N. R. and Das, J. (2004). A novel approach to design classifiers using genetic programming. *IEEE Transactions on Evolutionary Computation*, 8(2):183–196.

Nordin, P. (1994). A compiling genetic programming system that directly manipulates the machine code. In Kinnear Jr., K. E., editor, *Advances in Genetic Programming*, pages 311–331, MIT Press.

Nordin, P., Hoffmann, F., Francone, F. D., Brameier, M., and Banzhaf, W. (1999). AIM-GP and parallelism. In *Proceedings of IEEE 1999 Congress on Evolutionary Computation*, pages 1059–1066, IEEE Press.

Nordin, P., Banzhaf, W., and Francone, F. D. (1999a). Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Spector, L. et al., editors, *Advances in Genetic Programming Volume 3*, pages 275–299, MIT Press.

O'Neill, M. and Ryan, C. (1999). Evolving multi-line compilable c programs. In Poli, R., Nordin, P., Langdon, W. B., and Fogarty, T. C., editors, *Proceedings of the 2nd European Workshop on Genetic Programming*, pages 83–92, Springer-Verlag.

Perkis, T. (1994). Stack-based genetic programming. In *Proceedings of the first IEEE International Conference on Evolutionary Computation*, pages 148–153, IEEE Press.

Poli, R. (1999). Parallel distributed genetic programming. In Come, D., Dorigo, M., and Glover, F., editors, *New Ideas in Optimization*, Section 6, McGraw-Hill.

Potter, M. A. (1997). *The Design and Analysis of a Computational Model of Cooperative Coevolution*. PhD Thesis, George Mason University.

Potter, M. A. and De Jong, K. A. (2000). Cooperative Coevolution: An architecture for evolving coadapted subcomponenets. *Evolutionary Computation Journal*, 8(1):1–29.

Ryan, C. (2000). *Automatic Re-Engineering of Software Using Genetic Programming*, Kluwer Academic.

Ryan, C. and Ivan, L. (2000). Paragen – the first results. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P., and Fogarty, T. C., editors, *Proceedings of the Third European Conference on Genetic Programming*, pages 338–348, Springer-Verlag.

Stoffel, K. and Spector, L. (1996). High-performance, parallel, stack-based genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 224–229, MIT Press.

Svangård, N., Nordin, P., Lloyd, S., and Wihlborg, C. (2002). Evolving short-term trading strategies using genetic programming. In *Proceedings of IEEE 2002 Congress on Evolutionary Computation*, pages 2006–2010, IEEE Press.

Teller, A. and Veloso, M. (1996). Pado: a new learning architecture for object recognition. In *Symbolic Visual Learning*, pages 81–116, Oxford University Press.

Tomassini, M. (1999). Parallel and distributed evolutionary algorithms: a review. In Neittaanmki, P. et al., editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 113–133, Wiley.

Trenaman, A. (1999). *The Evolution of Autonomous Agents Using Concurrent Genetic Programming*. PhD Thesis, National University of Ireland.

Walker, J. A. and Miller, J. F. (2004). Evolution and acquisition of modules in cartesian genetic programming. In Keijzer, M., O'Reilly, U., Lucas, S. M., Costa, E., and Soule, T., editors, *Proceedings of the Seventh European Conference on Genetic Programming*, pages 187–197, Springer-Verlag.

Wong, M. L. and Leung, K. S. (2000). *Data Mining Using Grammar Based Genetic Programming and the Applications*, Kluwer Academic.