Generic systolic array for genetic algorithms

A Generic Systolic Array for Genetic Algorithms [1]

G. M. Megson and I. M. Bland
Parallel, Emergent and Distributed Architectures Laboratory,
(PEDAL),
Algorithm Engineering Research Group,
Dept. of Computer Science,
University of Reading,
Whiteknights,
P.O. Box 225,
Reading RG6 6AY.

May 1996

[1]Paper in IEE Proceeding Computers and Digital Techniques, Vol144(2):107-121, March 1997

Abstract

Genetic algorithms (GAs) are now established search and optimization techniques employed in a diverse range of applications. The run-time and hence quality of solutions produced by GAs is related to the choice of genetic operators, the size of the population, the application, and the manner in which the algorithm is implemented. Recently there has been considerable activity in accelerating GA performance by exploiting the explicit parallelism of the evolutionary mechanism both on general purpose parallel architectures and more recently on special purpose devices via rapid prototyping admitted by the availability of Field Programmable Gate Arrays (FPGAs). Previous work has concentrated on migration models using partitions of the population on programmable processors and/or hardware implementation of the fitness function. In this paper we present a systolic design for a simple GA mechanism which provides high throughput and unidirectional pipelining by exploiting the inherent parallelism in the genetic operators. The design computes in $O(N+G)$ time steps using $O(N^2)$ cells where N is the population size and G is the chromosome length. The area of the device is independent of the chromosome length and so can be easily scaled by replicating the arrays or by employing fine grain migration. The array is generic in the sense that it does not rely on the fitness function and can be used as a accelerator for any GA application using uniform crossover between pairs of chromosomes. The design can also be used in hybrid systems as an add-on to compliment existing designs and methods for fitness function acceleration and island style population management.

1 Introduction

Genetic algorithms (GAs) are search mechanisms that embody the principles of natural selection to develop solutions to a wide range of search and optimization problems. GAs have proven particularly useful for problems where effective individual heuristic techniques are not known or where the search space is noisy or has multi-modal properties. GAs were first presented formally by Holland [1], they operate by maintaining and manipulating a population of candidate solutions, called *chromosomes*. The chromosomes are constructed from an alphabet of symbols which are used to encode trial solutions to the problem at hand. In keeping with the genetic analogy, each individual symbol is referred to as a *gene*. In the canonical genetic algorithm a 2-ary alphabet is used and the chromosomes are simple binary strings. However it is not uncommon for more complex encoding schemes employing gene alphabets of a higher cardinality to be used. Research in the literature indicates that the coding scheme should represent the problem as closely as possible. Many problems, which previously have been tackled using binary encoding schemes have benefited from being re-encoded with a more natural, non-binary scheme (e.g. Job Shop Scheduling [2, 3, 4]). Conversely when binary is the natural representation of the problem such an encoding performs very well (e.g. [5]). However non-binary encoding schemes are rather artificial and have a limited meaning when hardware is considered. This follows because all encoding schemes are ultimately reduced to binary representations. The concept of non-binary schemes is embodied simply by gronping bits together into atomic units which cannot be disturbed by the action of the algorithm. Consequently we shall concentrate on binary representations.

In a GA each chromosome has an associated *fitness* value which is a qualitative measure of how good the chromosome is as a solution to the problem at hand. The fitness value is used to bias (or direct) the stochastic selection of chromosomes (normally pairs or mates) which are then used to generate new candidate solutions through a process termed *reproduction*. The new chromosomes replace the old ones in the population and this process is iterated over a number of *generations* until a solution is found or an acceptable degree of optimization is obtained. Reproduction consists of two operations, *crossover* and *mutation*. Crossover generates new chromosomes by combining sections of two or more selected parents. In simple One-point crossover, two parent chromosomes are cut at a random site and recombined to form a new child chromosome by swapping the tail sections of each chromosome. In practice more elaborate crossover strategies are often used for example n-ary point which cuts the chromosome into $n + 1$ pieces. n-ary crossover can be extended to its logical conclusion so that there exists a crossing point between each gene and the new child is constructed by making a decision *for each gene* which parent it is to come from. The decision is made by generating a random binary string (or template) and choosing a gene from parent 1 if the bit at the particular position is one, or from parent 2 if the bit is zero. The second child is constructed in a similar manner with the roles of the bits reversed. Such a strategy is called Uniform Crossover and is generally favored over other general crossover schemes in the belief that it minimises disturbance of the good genetic material or schema which occurs as a result of crossover [6]. A good general discussion on Uniform Crossover and its relative merits over other crossover schemes can be found in Beasley et. al. [7[. Which type of crossover mechanism is adopted is largely problem dependent, although it can be argued that the more chromosomes involved the more information you have to characterize good solutions. Clearly there is a trade-off between the computational complexity of the operator and the observed effects in the evolving population but to date no general results on how to exploit this correspondence have appeared in the literature. We shall concentrate on using Uniform Crossover between two selected parents to produce two children. Mutation acts on individual genes by randomly selecting genes which are then altered. For the canonical genetic algorithm with binary alphabet this is simply a case of flipping the state of a binary digit. In general the process is complicated by the need to ensure that genes mutate into valid forms in the alphabet. Mutation is used to ensure diversity. For example, when the GA begins to converge towards a solution the population tends to separate into groups of duplicate chromosomes. In some cases the optimal solution chromosome may not be derivable purely from the application of crossover. Mutation prevents such sub-optimal solutions from persisting and thus dominating the algorithm.

The fundamental aim in developing a hardware GA is to speed up the execution of the algorithm. Accelerated performance allows larger populations to be processed for more generations, hopefully yielding an improvement in the quality of solutions for the same computation time. We say hopefully, because it is possible that the greater number of iterations may give rise to some unsuspected degenerative effects that produce no improvements to the solution. Alternatively, if significant performance enhancements can be obtained, it becomes feasible to consider

applying genetic algorithms to non-trivial real-time problems such as robot path planning [8]. For such dynamic problems there is also a desire to embed the algorithm into larger hardware/software systems. A dedicated GA 'co-processor' seems a logical way to achieve this goal. The speed-up offered by hardware also admits the possibility of employing GAs to problems where the search spaces are so large that traditional methods are regarded as impractical.

There are a number of recent designs for hardware genetic algorithms [9, 10, 11, 12, 13, 14, 15] but these are mostly based on the following premise. It is estimated that a sequential GA spends on average around 65 % of the total processing time on fitness evaluation [9]. Consequently there would seem to be great incentive for performing a number of function evaluations in parallel, or migrating the fitness function into hardware, and less incentive for parallelizing the genetic operators. In this paper we argue that this approach, albeit not mis-guided, is somewhat naive. This argument is based on the following observations. Firstly, the evaluation of the fitness function is often a complex procedure which is not easily mapped into hardware, dictating against specialist fitness engines or a least to hardware with limited applicability. Secondly, given a distributed 'high level' parallelization, where the fitness functions of chromosomes are computed in parallel batches with migration to preserve diversity, the real bottleneck becomes the genetic operators. A simple approach would be to implement two orthogonal levels of parallelism at the population and operator levels. However this implies architectures capable of exploiting both medium and low level granularity with equal effectiveness. We propose that this can be achieved by using specialist hardware for the low granularity genetic operators, combined with more general processors for the higher levels. Below we describe the architecture for implementing the genetic operators in hardware as systolic arrays. These arrays are generic in that they apply to all algorithms using Uniform Crossover. Similar principles can be used for more algorithmically specialised operators. The systolic approach is very well suited to VLSI implementation and can be synthesized by the application of well understood algorithm engineering principles [16] based on loop unrollings of the underlining genetic algorithm. In addition we propose a fixed sized solution where the array size is independent of the population size or chromosome length. Where the fitness function is particularly regular in structure (e.g associative operators) it may be possible to incorporate our arrays into embedded systems with hardware based fitness evaluation.

The systolic array design methodology has traditionally been used to design special-purpose VLSI devices. Although systolic designs are generally aimed at VLSI, we are mindful to remember the benefits offered by using FPGAs as the implementation technology instead. FPGAs not only provide a means of rapid prototyping of VLSI designs but many devices offer the advantage of reconfiguration. With the range of genetic operators that can be used in GAs, reconfiguration allows alternative operators to be developed and 'plugged in'. We therefore shall consider both VLSI and FPGAs implementations when considering our systolic design.

The rest of the paper is organised as follows. Section 2 presents the systolic array genetic algorithm in detail including cell definitions for the array elements. In Section 3 we perform an area-time complexity analysis of the design and compare this with the serial case. Section 4 explores the ability to scale the arrays to improve the genetic performance. Conclusions are presented in Section 5.

2 Systolic Array Genetic Algorithm

Moldovan [17] points out 'VLSI is a raw technological environment which requires new ideas in computer organization, theory of computing and other related fields'. Systolic arrays were proposed by Kung [18, 19] and others in an attempt to address some of the problems associated with the design of special-purpose systems in VLSI. The systolic array principle is to distribute the computation over a number of computational units which are locally connected to form an array. Each unit or cell contains simple computational logic with a few registers acting as local memory. Data is fed through the array from the boundary cells and 'pumped' through the cells in a pipelined fashion (the analogy between pumping data through a array and pumping blood through the body explains why the array is termed 'systolic'). As data is only fetched from memory at the boundary cells, it is re-used extensively within the array. The pipeline exploits concurrency, which in combination with data re-use greatly reduces the overall execution time of the algorithm. Many systolic arrays exploit the pipelined principle in more than one direction, resulting in two and three dimensional array structures. The regular array structure and the simple computational functions of the cell results in a data-flow which is likewise simple and regular. This kind of data-flow has substantial advantages in terms of design and implementation over complicated ir-

regular communication. Primarily long communication lines can be almost completely avoided as each cell is connected locally. The array also results in devices which require much less effort in design and implementation. The nature of the array means the design is both modular and expandable and in combination with modern photo-lithographic processes they are comparatively easy (and therefore comparatively less expensive) to design and implement than traditional ASICs.

Designing systolic algorithms is principally concerned with identifying the data and control dependencies which exist in an algorithm. These dependencies define the order in which tasks have to be performed and the absence of dependencies indicates places in the algorithm where parallelism can be exploited. The dependencies can be expressed mathematically, as a series of recurrence relations which can be used to define an array structure to implement the algorithm through the process of synthesis. Work combining systolic arrays with genetic algorithms can be found in Chan and Mazumder [9]. They propose a systolic array to assess the fitness for the hypergraph partitioning problem. We see this work as being of significance although we are concentrating on implementing the genetic operators (rather than the fitness function) for the reasons given above.

The systolic genetic algorithm we propose is based around a number of systolic arrays arranged as a macro-pipeline as shown in Fig 1a. An immediate advantage of using the systolic array approach is that we can scale the array in accordance with the population size and use the concept of pipelining and data re-use to 'pump' the chromosomes through the array. The rationale behind this decision is that we can implement populations as a number of sub-populations (usually referred to as demes) and use the coarse-grained parallel genetic algorithm model [20] in a locally parallel, globally sequential [21] mode. Such a model would allow each deme to be either processed serially through the device or for the whole design to be replicated a number of times over the silicon area and realize a true coarse-grained parallel genetic algorithm. The size of each deme can be chosen arbitrarily, either in relation to GA theory [22] or depending on the amount of silicon resource available (this is particularly applicable to FPGA implementations where gate density is rapidly being improved). The design follows from a particular projection of the un-rolled data dependence graphs of the various genetic operators such that a) they are mutually compatible for uni-directional pipelining, and b) that the array is independent of the chromosome length. Such a design is not at all obvious from the usual GA specifications. Consequently we can construct hardware without having to use wide data registers which would require sophisticated control logic to accommodate varying lengths of chromosomes. We also do not need to resort to distributing the optimization of the chromosome over a number of fixed length sections (as in [14, 15]), which may not take into account the interaction (or *epistasis*) of the genes. Uni-directional pipelining assists in designing a fault-tolerant device [23]. It also simplifies the design of extensions to the pipeline such as alternative genetic operators.

2.1 The Macro - Pipeline

The array itself is arranged as a macro pipeline of seven distinct functional units pipelined together. The seven arrays are as shown in Figure 1.

Each of the seven systolic arrays performs some manipulation of the chromosomes as they pass through along the horizontal axis of the array. Crossover and mutation are each achieved in a single systolic array while selection, which in this case is roulette wheel selection [24], is distributed over the remaining five arrays. Selection is achieved in rather an idiosyncratic way and may require some moments of contemplation to confirm that the principles of selection are preserved. This feature arises as a direct result of manipulating the data dependencies of a genetic algorithm to fashion the arrays and indicates that a detailed study of the algorithm at this level can generate designs which traditional techniques may not consider. Control data which is required by the array is passed down the vertical axis and skewed to meet the chromosomes in the correct cell. Some pre-loading of registers can be achieved before the chromosomes have propagated through the previous arrays in the design but in most cases timing is achieved by staggering the entry times of the chromosomes. This stagger is set up before the chromosomes enter the first array and is preserved by all the arrays in the design. Consequently the skewing (and deskewing) can be achieved simply by pre- (post-) arrays of delay cells if required.

An essential part of any genetic method is the generation of random numbers, used to meet the stochastic requirements of the algorithm. As part of the design processes we investigated the data dependencies involved in random number generation [25]. We have used the mixed congruential Pseudo Random Number Generator (PRNG)[26] which, conveniently, already exists in the form of a recurrence relation, $x_{n+1} \equiv \lambda x_n + \mu$. On a sequential processor we would iterate this recurrence and so generate a stream of numbers, singularly, at each

Figure 1: (a) The Macro Pipeline (b) Structure of the chromosome

successive time step. To achieve a parallel RNG we have simply un-rolled the loop into an array and fed the top cell with the output of a free-running PRNG (which uses a *different* recurrence relation). Each cell receives as its seed the output of the preceding cell to generate a new random number. At the same time the cell passes out the result of its last calculation to the next cell to act as its seed. As time passes each cell generates (concurrently with the other cells) its own stream of random numbers. What is significant about this approach is that we are not left with isolated PRNGs which might, due to the deterministic nature of the generator and poor seeding, produce streams of numbers which are significantly similar. As each cell is effectively re-seeded each time step, the generators follow the top RNG. Each cell disturbs the stream generated above (randomly) to produce a new independent stream below. It is true that, under certain circumstances, the random numbers generated may begin to repeat but this repetition is distributed across all of the cells and so does not affect the independence of the streams generated with each cell in respect both to themselves and all other streams in the array. This design can be seamlessly incorporated into the arrays (e.g. Mutation, see Fig 9) without having to use long communication lines from a central PRNG.

A chromosome is structured as illustrated in Fig 1b. Preceding the 'genetic' portion of the chromosome are the two fields, fitness and select. Fitness holds the fitness value of the chromosome, derived from some fitness evaluation routine. This arrangement divorces the design from any problem-specific fitness function. The Select field is used internally as a non-stationary scratch register, primarily to store the number of times (if any) a particular chromosome has been selected for reproduction, hence its name. The five selection arrays primarily deal with these fields and set up paths for the 'genetic' portion of the chromosome to follow through the array. For clarity of description both of these values can be regarded as whole numbers which are communicated in parallel. In the finished design, data will be passed between cells either bit or byte-serially. The genetic portion of the chromosome follows these values bit-serially through the arrays. It is this section of the chromosome which is manipulated by the genetic operators.

The recurrence relations we have identified are given below in the form of systolic array cell definitions. A diagram of each array accompanies these definitions as well as a description of the functionality of the particular arrays. The fitness and select registers will be referred to as F and S respectively in the cell definitions. The relation $X_{in} \in F$ could be read ' X_{in} is a fitness value'.

Figure 2: The Roulette Wheel Array with a snapshot of of the data after four time steps

2.1.1 Roulette Wheel

Roulette wheel selection is based on the idea that each chromosome occupies a section of a roulette wheel. The size of each section is proportional to the fitness of each chromosome in terms of the percentage fitness w.r.t the total fitness of the population. The wheel is 'spun' (which corresponds to the selection of a random number between 0 and 1) and the section where the hypothetical ball lands indicates the chromosome to be chosen for reproduction. As the size of each section of the wheel is proportional to the fitness of the chromosome it represents, there is a bias towards selecting fit chromosomes but it is important to note that selection is achieved stochastically. In the array (Fig 2) the roulette wheel is formed as follows. The array takes the fitness value from the first chromosome which enters at the front of the array at the top left cell. A total is passed vertically downward to meet the fitness value of the remaining chromosomes on successive steps. Thus as the data moves downwards all the fitness values are summed and the total fitness for the population or deme appears at the bottom of the column. This sum is then used to scale the fitness value by synchronising with the chromosomes which have been propagated horizontally through a batch of delay elements. The fitness value is overwritten by replacing its percentage value w.r.t. the total fitness of the population (or deme). Clearly the population size (N) is required to achieve this calculation and can be passed into the array from the top or implicitly assumed by the array cells (which requires pre-loading). The select value and chromosome fields following the fitness value through the array remain unaffected. Fig 2 also shows the position of the data after four time steps. The format of the chromosomes can be clearly seen with the Fitness and Select fields preceding the actual chromosome (passed bit serially). Also visible is the stagger that has been put onto the population with the top chromosome entering the array first and each successive chromosome being delayed by one time step. This stagger is preserved by all of the arrays in the design.

The figure and cell definitions given above describe the roulette wheel array in an intuitive form. However this design requires a long data line to communicate the sum of fitnesses (from the leftmost cells) to the fitness scaling cells (rightmost cells). Such a line compromises the systolic principle of local communication between cells. We can eliminate this non local communication line if we adopt the technique of space folding [16]. The principle of space folding is to re-map two elements (in space) onto one location. Fig 3 gives the new array design and resulting cell definitions. The axis of the fold runs diagonally from top left to bottom right in the old array, including the fitness summing cells but excluding the scaling cells on the far right. The fold acts to re-map two delay cells onto one location in space. The fitness summing cells are included in the fold as these cells are implicitly used to delay the chromosomes. Folding the array results in the summing and scaling cells being placed adjacent to each other and so now only require a local communication line between them. A further advantage of this fold is to reduce the number of delay cells required, as each delay cell now operates in two dimensions. However it is also important to note that this has increased the complexity of these cells. This is especially true of the fitness summing cells which now also act as delay cells in the Y direction.

Figure 3: The Folded Roulette Wheel Array

2.1.2 String Selector

This stage actually performs the roulette wheel selection. A random number (between 1 and 100) is passed in to the array (Fig 4) representing the landing place of the 'ball' on the wheel.

As the 'ball' value moves vertically down the array the scaled fitness of each chromosome is subtracted. The cell where the sum becomes negative increments the following Select field of the chromosome which is then regarded as being selected. The principle of the roulette wheel is observed as highly fit chromosomes with accordingly higher scaled fitness have more chance of reducing the number below zero and thus being selected. Once the 'ball' value becomes negative it will remain negative for any further subtractions so the sign-bit of the subtractor can be used to increment the selection field. In the diagram the array has a width of N cells. As each column corresponds to a spin of the roulette wheel N selections are made, guaranteeing enough chromosomes to replace the current population after crossover. Note that it is important to realize that there is no limit to how many times a particular chromosome can be selected for reproduction (up to a maximum of N). Consequently some chromosomes may emerge from the array with the select field still zero. As before, we assume that the remaining chromosome, following the select field, passes through the cells unaffected. This is easily achieved by a control bit tagged to the random numbers pumped down the columns (for example making the sign bit negative before entering the array).

2.1.3 String Sort

In this array (Fig 5) the chromosomes are sorted by select field with the most selected chromosome emerging from the bottom left cell. By analysing the recurrences, strictly speaking it is not necessary to sort the chromosomes at this point. However the regularity and simplicity of the rest of the design far outweighs the additional cells and latency introduced. The sort algorithm is a standard systolic version of bubble-sort [16] modified to take into account the trailing 'genetic' bits. Thus the 2-D dependence graph cannot be projected into 1-D to produce a standard 1-D array for sorting. The array is derived by unrolling the associated two loop program which compares elements i and j in a list with $i = 1, 2, .., N-1$ and $j = i+1, ..., N$ hence the triangular shape. Chromosome strings enter the array horizontally from the left with the select field acting as the sort key. The preceding fitness fields will be detached from the associated strings as a result of exchanges but this poses no problems as we will not use its value again in the rest of the macro-pipeline. Essentially a chromosome string hitting a diagonal cell is propagated down the associated column where it encounters other strings. If we consider the i th column, this will

Figure 4: The String Selector Array

Figure 5: The String Sort Array

Figure 6: The String Match (Stage 1) Array

route the i th string downwards to be compared with the strings $i+1, ...n$. Thus column i of the array performs the comparisons of string pairs $(i, i+1)$, $(i, i+2)$, ... ,(i, N). Clearly with N columns all the comparisons of the bubble-sort algorithm are performed. Next we have to convince ourselves that we can exchange two strings i and j without corrupting the data following the select field. To do this, observe that cells act as comparators and routers. When two select fields enter a cell they are compared. If $select_i < select_j$ we have to swap the data paths of the two strings. This is achieved by setting a switch and then disabling the comparator to allow trailing bits to be routed correctly. Alternatively we can tag a control bit to the select field so the comparator is enabled only when two valid select fields are present in the cell. Clearly this organisation ensures that a cell only performs on valid comparisons and that the correct routing paths develop behind a wavefront of comparisons moving across the array from top left to bottom right.

2.1.4 String Match (Stage 1)

The next step in the algorithm is to mate the genes. From an hardware point of view it would be simpler if we could guarantee that mates were next to each other in the array. Otherwise a fully connected interconnection scheme will be required to mate any string i to any other string j as dictated by the probabilistic nature of the GA. For simplicity and regularity of design we break this mating step into two matching stages. The first stage of string matching replicates the chromosomes depending on how many times they have been selected. The second stage randomly mixes the strings to ensure some diversity in the mating process.

In stage 1, if a string has been selected four times (i.e its select field is set to four), four copies of the string will appear at the outputs (on the right hand side) of the array (Fig 6). The array works as follows. Strings enter the array from the left in skewed format and move horizontally. Now consider a particular column j and suppose that $select_1 > 1$ and $select_i = 1$, for strings $i = 2, ..N$. As string 1 enters cell $(1, j)$ (numbered from top left) the select field is examined (recall that the fitness field is not in use at this stage). Because the select value is greater than one, we need to make a new copy of string 1. This is achieved by decrementing the select field and passing the string onto column $j+1$ and also routing a copy of the string vertically to row two of the array, but with the select field set to one. Now consider cell $(2, j)$ on the next time step. String 2 enters from the left and the replicated string 1 from above. Thus string 2 must be moved down and the new string 1 moved right. This procedure is repeated for all the cells $i = 2, ..., N$ in column j so that rows 1 and 2 of the array carry copies of string 1 with select fields $select_1 - 1$ and 1 respectively and row i carries string $i-1$ from columns j to $j+1$.

Observe that string N was lost.

The above scenario does not define the complete functionality of the cells. Suppose we define p_{ij} as the string number entering the cell (i,j). Associated with this string is the value of its select field which we shall denote as $select_{p_{ij}}$. If no string is present at the input to the cell we shall assign a value to this field such that $select_{p_{ij}} = -1$. Now consider cell (i,j) of the array. This cell receives input from the left and top and produces output from the right and bottom. In particular the cell receives string $p_{i-1,j}$ from the top and string $p_{i,j-1}$ from the left. Now, when $select_{p_{i-1,j}} = -1$ we can assume that there is no string arriving from above and proceed to examine $select_{p_{i,j-1}}$. If the value is greater than one we need to replicate string $p_{i,j-1}$ as previously so that string $p_{ij} = p_{i,j-1}$ and $select_{p_{ij}} = select_{p_{i,j-1}} - 1$ and $select_{p_{i+1,j}} = 1$. The case when $select_{p_{i-1,j}} \geq 0$ indicates a string has been pushed down from cell $(i-1,j)$ and the string needs to takeover row i. Thus $p_{ij} = p_{i-1,j}$, $select_{p_{i,j}} = select_{p_{i-1,j}}$ and $select_{p_{i+1,j}} = select_{p_{i,j-1}}$. It may well be the case that $p_{i-1,j}$ is a zero selected string but we aim to percolate these strings through the design so that they can be removed from the array at the bottom boundary. For the case where $select_{p_{i-1,j}} = -1$ and $select_{p_{i,j-1}} = 0$ *or* 1, no replication occurs in cell (i,j). By virtue of the skew on the inputs to column j the cells (i,j) are operated in the sequence $i = 1,..,N$. Thus if string p_{ij} is replicated, all strings p_{rj} where $r = i+1,...,N$ will be shifted down one row with any string requiring duplication being shifted rather than replicated. As a result of this shift, string p_{Nj} is shifted off the bottom of the array and lost. It also follows that if string p_{ij} is replicated, all strings p_{rj} where $r = 1,..,i-1$ have select fields set to one and therefore no replications have occurred to these strings in column j. If we set $select_{p_{0j}} = -1$ to force correct behaviour at the boundary of the array each column can replicate at most one string. Thus $N-1$ columns are required to replicate N strings and, as each column can shift only one string off the bottom, we lose at most $N-1$ strings. Indeed it is easy to see that the array satisfies an invariant such that

$$S_j = \sum_{i=1}^{N} select_{p_{ij}} = N \quad and \quad S_j = S_{j-1} \quad for \quad j = 1,...,N-1$$

from which it follows that if we are to avoid losing any selected strings, input to the array should be ordered according to $select_i \geq select_{i+1}$ for $i = 1,...,N$. That is, strings must be sorted before entering the array with the most selected string entering row one and the least selected string entering row N. If any string is selected more than once the invariant implies that there must be a zero selected string in the bottom row of the array so that shifting to replicate the string causes no loss of useful information. This also justifies the use of the sorting array described above.

The cell definitions given express the operation of the array in terms of one of three particular states the cells can adopt. By virtue of the skew on the incoming chromosomes and by the fact that each cell preserves this skew by passing strings to the left, a wavefront is established which moves diagonally through the array starting at cell $(1,1)$. In combination with this wavefront and by remembering which particular state a cell adopts, the trailing chromosomes can pass through the array and be subject to replication where desired.

2.1.5 String Match (Stage 2)

The chromosomes emerging from the stage one array need to be paired for crossover. To preserve the notion of local communication of data within the arrays, it would be advantageous to mate pairs of chromosomes which are physically next to each other. Unfortunately a block of strings replicated $r > 0$ times will be emerging in a contiguous block of $r+1$ strings. Applying crossover to a group of these strings within the block has no effect because identical genetic material is exchanged. Thus the purpose of the stage 2 array (Fig 7) is to randomly shuffle the chromosomes so that effective crossover can occur between neighbouring chromosomes. Essentially the shuffling is achieved by overwriting the fitness field (no longer in use) with a random value between 1 and N inclusive and then sorting the strings using the random values as a keys. The array consists of two parts. The first part is the first column of the array, which is just a linear unrolling of the PRNG recurrence where the cells are augmented with control signals to overwrite the select field of a string, this leaves the trailing 'genetic' information unchanged. Allowing for the skew of the input data a simple tag bit added to the input seed at the top of the column will provide the correct activation as it moves down the column. The second part of the array is a copy of the sorter discussed above.

Figure 7: The String Match (Stage 2) Array

2.1.6 Crossover

The next step is to actually mate the selected strings. Following the previous discussion, mating is achieved by applying the standard crossover operator to pairs of strings entering odd and even numbered rows of the array (Fig 8). In practical terms this requires a single column of $N/2$ cells with two inputs and two outputs. The string on the odd numbered row is delayed by one step to align the two strings before entering a switch unit, then the even numbered string is delayed to restore the skew before the strings leave the cell. The switching unit can be controlled in one of two ways. In the first scheme the vertical connections carry a stream of bits. A '1' bit entering the cell toggles the output paths while a '0' bit allows the string to pass through unchanged. If we imagine the vertical input to be a vector of N bits any crossover operator on two strings can be defined by setting appropriate elements of the vector to zero or one. For example a one-point crossover at any position $N \geq r > 0$ would require a one bit in the r th position of the vector with the rest all zero (i.e and elemental vector). Similarly two-point crossover at positions r and q would require one bits only at positions r and q. Uniform Crossover would use all of the bits in this vector to repeatedly cross over the two streams. A possible drawback to this scheme is that the same crossover is applied to all pairs of strings as the bit-vector filters down the column of cells. A more sophisticated method involves random numbers generated at each cell.

In this scheme we overlay an unrolled PRNG on the column of cells. A (seed) random number is entered vertically and is used to generate random numbers in all of the crossover cells. The seed value is used to calculate a new random number (using the Mixed Congruential recurrence) which is passed to the next cell to act as a seed. The result of this is a random number generated in each cell at each step. For n-ary point crossover we need to store n random numbers and compare them with the length of the chromosome (which we can also pass down the array) to determine where the crossover points occur. This method is suitable for one or two point crossover. However the scheme becomes cumbersome for large values of n as n registers are required to store these random numbers. The scheme really comes into its own when it is used to implement Uniform Crossover. Uniform Crossover makes a choice between parents for every gene in the child and this can be achieved in a bit-serial fashion. The advantage here is that we do not need to store the crossover sites. The random numbers can be used and discarded every time step removing the need to pass the length of the chromosomes down the array. We have discussed the effectiveness of this scheme in comparison with other, general, crossover strategies and therefore Uniform Crossover is the logical choice for implementation. This is reflected in the figure which contains cell definitions for a Uniform Crossover operator.

Figure 8: The Crossover Array (Uniform Crossover)

2.1.7 Mutation

The final stage of the pipeline provides some random perturbation of the bits of the chromosome. The array (Fig 9) is trivial, being just a column of N cells again based on the unrolled form of the PRNG. A (seed) random number is entered into the topmost cell; successive cells then generate a new random number down to the next cell. The decision whether to mutate or not is taken by comparing a random number with the mutation probability P_{mut}. This is passed to all mutation cells and is stored by the cell to be used as a measure of whether to mutate or not. The value of P_{mut} can be varied depending on the problem being tackled or dynamically as the population begins to converge towards a solution. The systolic random number generator described above ensures that each mutation cell generates its own independent stream of random numbers to ensure no biasing factors can affect the independence of the mutation. A comparison between the random number and P_{mut} is made as every gene passes through the cell. If the random number is smaller than P_{mut} then mutation occurs, otherwise the gene carries on through the cell unaffected.

3 Area-Time Complexity

Assuming the population is size N, the area.time complexity of each array is as follows:

In the Roulette wheel array (Fig 3) we have a triangular array in which the first column computes the sum, the bottom row scales the fitness and the rest of the array routes data. The first chromosome enters the array one cycle on and percolates down the column in N steps to emerge from the array after $N+1$ steps. Subsequent chromosomes strings are skewed by $i-1$ cycles for the ith row so the last string starts to enter the array after N cycles and takes $N-1$ steps to reach the scaling cells. If we let G be the length of the strings (including the fitness and select fields) the array completes computations after T_1 cycles and has A_1 cells where

$$T_1 = 2N + G + 1 \qquad A_1 = \sum_{i=1}^{N} i + N = (\frac{N}{2})(N+1) + N = \frac{N^2}{2} + \frac{3N}{2} \tag{4.1}$$

In A_1 the first term is the number of cells in the triangular part of the array and the second the size of the scaling column.

Figure 9: The Mutation Array

The selection array (Fig 4) uses a column of cells to make one selection. Since we want to make at most N selections so that the next generation has the same number of strings we require N columns. It follows that the delay through the array is N cycles so that the first row of the array outputs data after N cycles, the last row starts output after $2N$ cycles and the complete selection process finishes after T_2 cycles. The array is a square array having A_2 cells where

$$T_2 = 2N + G \qquad A_2 = \sum_{i=1}^{N} \sum_{j=1}^{N} 1 = N^2 \tag{4.2}$$

The string sorter is a standard triangular array for sorting lists of N items (Fig 5). The first string enters the array on the first cycle and filters down the first column to emerge on step $N+1$. The last string enters the bottom row of the array on cycle N and moves horizontally for $N-1$ steps producing a result after $2N$ steps (the time for sorting a list of integers). Because we are dealing with strings rather than single numbers the sort array finishes after T_3 cycles. The array has A_3 cells.

$$T_3 = 2N + G \qquad A_3 = \sum_{i=1}^{N} i = (\frac{N}{2})(N+1) = \frac{N^2}{2} + \frac{N}{2} \tag{4.3}$$

The Matching arrays consist of two parts. The first stage (Fig 6) is restricted form of router in which strings can only be moved down. Although the exact data flow of the strings depends on the problem instance (in fact the random number used in selection) we observe that the input/output format is identical and that the skew is preserved from column to column in the array. Consequently the delay through the array is N cycles and allowing for the skew gives a total output time of T_4. The number of cells (A_4) is equivalent to A_2.

$$T_4 = 2N + G \qquad A_4 = A_2 = N^2 \tag{4.4}$$

The second stage (Fig 7) is to randomly mix the population ready for mating. This is achieved by a single column of cells which assigns a random value to the select field with once cycle delay and N cells followed by a sorting array identical to the one above with time $2N + G$ and A_3 cells so

$$T_5 = T_3 + 1 = 2N + G + 1 \qquad A_5 = A_3 + N = (\frac{N}{2})(N+1) = \frac{N^2}{2} + \frac{3N}{2} \tag{4.5}$$

The next stage of the array involves the actual crossover (Fig 8) which requires only a single column of $\frac{N}{2}$ macro cells (each cell equivalent to one cell in the rest of the design). During crossover the skew between odd and even strings is removed before and then restored after application of the operator. Consequently the delay through the cell is two cycles. Thus crossover is complete after

$$T_6 = N + G + 2 \qquad A_6 = \frac{N}{2} \tag{4.6}$$

Finally, Mutation can be achieved at each step as the chromosome passes through the mutation array (Fig 9). Mutation therefore has a time complexity of one cycle and N cells. That is

$$T_7 = 1 \qquad A_7 = N \tag{4.7}$$

Putting the various component parts together we find a total delay through the array a follows. The values of T_1 to T_7 give the total time for a string to fully pass through the arrays. The total time for the pipeline is the summation of these times, however as the arrays are pipelined, the next array in the pipeline receives input as the first chromosome emerges and therefore the $N + G$ steps required to fully empty the previous array occur concurrently with the first $N + G$ steps of the following array. In the calculation of the total time we need to adjust the figures T_1 to T_7 to reflect this. This can be achieved by removing a factor of $N + G$ from each T_1 to T_7. So that the total time for the whole pipeline becomes.

$$T = 5N + 5 \tag{4.8}$$

We still need to reflect the stagger of N steps before the Nth chromosome exits the pipeline and the further G steps required to fully empty the array. The total time required for the pipeline is therefore

$$T = 5N + 5 + N + G = 6N + 5 + G \tag{4.9}$$

The total number of cells is given simply by

$$A = A_1 + A_2 + A_3 + A_4 + A_5 = 3N^2 + \frac{11N}{2} \tag{4.10}$$

In terms of orders of complexity the Time complexity of the algorithm is $O(6N)$ and the Area complexity is $O(3N^2)$.

The sequential computation time can be evaluated as follows. We break down the GA into the three operations of selection, crossover and mutation and as the algorithm is being run sequentially we simply sum the times for each of these operations. In selection the first task is to calculate the total fitness of the population in order to proportion the roulette wheel. If we continue to assume we have N member in the population, this will take N steps. In a serial, software GA there is no need to physically scale each individual fitness by this figure as this can be achieved by scaling the range of the random number used to select slots on the wheel. Once this number is generated, the population is searched to find the chromosome which corresponds to the chosen section of the wheel. This will take on average $\frac{N}{2}$ steps to achieve assuming a linear search of the population (a binary search can be used here, see [27]). To fully replace the population, $N - 1$ further chromosomes need to be selected and so the total time taken in selection is

$$T_{select} = N + N(N/2) = \frac{N^2}{2} + N \tag{4.11}$$

Once a pair of chromosomes have been selected, they need to be subjected to crossover. In some GAs there is a stochastic probability that crossover between mates may not occur but for the sake of simplicity we shall assume that crossover occurs between all pairs of selected chromosomes. The time taken to achieve crossover (Uniform) is proportional to the number of genes in the chromosome, i.e. its length which we shall call L (not to be confused with G as used in our design as this includes the fitness and select registers in addition to the actual chromosome). We will assume that 1 time step is required to generate the random number required to select the parent gene and a further time step is required to copy the gene to each offspring. Therefore 3L time steps are required to produce the two children. This operation is repeated $\frac{N}{2}$ times to fully replace the generation.

The total time for crossover is therefore,

$$T_{crossover} = \frac{N(3L)}{2} \tag{4.12}$$

Mutation is achieved by making a bit-by-bit decision on whether or not to mutate. For each chromosome this takes L units of time where L is the number of genes in a chromosome. It is possible to use statistical method at the chromosome level to calculate which genes need to be mutated. However we shall ignore these methods in order to make a comparison with the hardware device. Every gene in every chromosome can potentially mutate so therefore the total time required to perform mutation is

$$T_{mutation} = LN \tag{4.13}$$

The total time in the serial case is therefore.

$$\frac{N^2}{2} + \frac{N(3L)}{2} + LN = \frac{N^2}{2} + \frac{5LN}{2} \tag{4.14}$$

Consequently the complexity of the sequential algorithm is $O(N^2)$. It is important to note the effect of using roulette wheel selection on both the serial and the systolic form of the algorithm. The serial algorithm's time complexity of $O(N^2)$ is as a direct result of using the scheme, which only selects one chromosome for every pass through the population. The systolic algorithm manages a more desirable $O(N)$ time complexity. However this is at the expense of an $O(N^2)$ area complexity which results from the need to have global access to the population. There are a number of ways to improve the performance of the roulette wheel selection scheme [28, 27]. It is also important to point out that roulette wheel selection, as well as other proportionate selection schemes are much slower than other techniques such as ranking or tournament selection [27]. Tournament selection is also a likely candidate for parallel implementation as it does not require global access to the population. We are using the Roulette Wheel scheme for two reasons, firstly it is a well understood scheme which is often used to describe the GA in the major texts on the subject. More importantly, Roulette Wheel selection preserves the *global* nature of the selection found in the original sequential algorithm. Tournament selection does not base its selection on the fitness relative to *all* the individuals in the population and in many cases a binary tournament is used where the fitness of only one other chromosome is known. It is as yet unclear what the effect of global versus local fitness knowledge is on the algorithm however recent research tends to be in favour of global selection schemes [29].

4 Scalability

One of our motives for using systolic arrays is to produce a design which can be scaled easily. We have deliberately designed the algorithm so that it can be scaled in accordance with the desired population size (allowing chromosome length to remain independent of the design) and at this stage it is important to assess the implications of this decision. The area/time complexity calculations above indicate that, although we achieve a considerable speed-up over the serial genetic algorithm, the area complexity of the design (particularly in the selection stages) is $O(N^2)$. Such a figure is perhaps un-surprising considering that we are using unrolled two- dimensional array structures and can be somewhat outweighed by the performance gains achieved by using massive parallelism and locally connected cells. However such an large increase in area usage with population will have a bearing on the size of the deme which can be implemented and this raises the question of how small demes will affect the *genetic* performance of the algorithm. Empirical evidence suggests that better overall solutions can be achieved by using multiple small populations rather than a single large one [30]. The benefit for our design in using multiple small populations can be expressed mathematically. If we were to double the size (N) of a particular population so that we have $2N$ chromosomes in total, doubling the size of a single population would involve $(2N)^2 = 4N^2$cells. However doubling the population size by the addition of N chromosomes in a separate population would involve $2(N)^2 = 2N^2$ cells, i.e. half the additional number of cells than for the single large population.

When considering scalability and the creation of a deme-based population, it is significant to note that our design allows the population to extend over a number of VLSI devices as a collection of demes. This allows problems which demand very large overall populations, which could not be accommodated on a single device,

Figure 10: (a) Trivially Parallel model using multiple devices. (b) Island Model using an Inter-connection Network

to be tackled. In order to achieve this an interconnection scheme needs to be defined which links the individual devices with the population members held in memory. Two examples of interconnection schemes are given in Fig 10. In the scheme depicted in Fig 10a each device is allocated a deme and processes the same deme over successive generations. In this model the demes remain independent from each other (A model which Graham and Nelson have called the *trivially parallel* model [11]). If we include a more sophisticated interconnection network, as in Fig 10b, we can achieve the goal of migration of chromosomes between demes and achieve a island based coarse-grained parallel genetic algorithm. In this scheme we can further impose migration schemes so that, for example chromosomes can only migrate to devices within a certain 'distance' from each other.

The exploitation of scalability also extends to the individual pipeline elements within the macro pipeline of Fig 1. We can scale certain arrays so that, in conjunction with the design's policy of replicating multiply selected chromosomes, we can employ some interesting population management strategies. Two examples are given in Fig 11. Additional columns have been added to the select array so more than N chromosomes are selected each generation. The Matching (Stage one) array has also been scaled to allow for these extra selections to be converted into actual copies of the chromosomes. In Fig 11a, only the top N chromosomes progress onto crossover and mutation, biasing these operators in favour of the fit chromosomes. In Fig 11b, the extension is continued to included the re-sizing of the matching stage 2, and crossover arrays so that the extended population progress through to crossover before being truncated. The top half of the chromosomes are then mutated and evaluated/returned to memory. Since the second matching array mixes the population prior to crossover, the effect on the emerging population is less biased towards the fit chromosomes. Instead the new population should include a higher proportion of fit schema. Of course such extensions will have an affect on both the area and time complexities of the algorithm. Suppose, for each extension, we wish to make an additional R selections. For the first extension, the chromosomes would require and additional $T_{Select} = R$ cycles to clear the select array, which would require $A_{Select} = NR$ additional cells. The Matching (stage 1) array would delay the chromosome by a further $T_{Match1} = R$ cycles and would require $A_{Match1} = (N+R)^2 - N^2 = R^2 + 2NR$ additional cells. The total additional time requires for the first extension becomes $T_{Total(a)} = 2R$ cycles, with $A_{Total(a)} = R^2 + 3NR$ additional cells. For the second extension the delay and cell count is increased further as a result of scaling the second Matching and Crossover arrays. $T_{Match2} = R$ cycles and $A_{Match2} = (N+R)^2 - N^2 = R^2 + 2NR$ additional cells are required for Matching (stage 2); $A_{Crossover} = \frac{NR}{2}$ additional cells are required in the crossover array, note that the delay through the scaled crossover array is the same as that for the un-scaled array. The total time and area increases associated with the second extension becomes $T_{Total(b)} = 3R$ cycles and $A_{Total(b)} = 2R^2 + 5NR + \frac{NR}{2}$

Figure 11: Extensions to the design achieved by scaling the individual pipeline elements (a) Fitter Chromosomes (b) Fitter schema

cells. The increases in number of cycles and cells required are outweighed if an improvement in the algorithms performance (i.e. convergence) can be obtained which reduces the size of the total population (i.e. the total in all of the demes) or requires fewer iterations (generations) to be processed.

5 Conclusions

The primary aim of this paper has been to accelerate the operation of the generic algorithm to reap the benefits of being able to tackle larger problems or reduce the processing time before an acceptable result is obtained. The systolic array genetic algorithm has been designed as a result of isolating the generic aspects of a genetic algorithm and exploring how these can be efficiently implemented in hardware. This is in contrast to most current research which concentrates on fitness function evaluation or production of specialized genetic operators. By concentrating on just the genetic operators we are able to design arrays which can accelerate these operators considerably. The task of fitness evaluation, which is by its nature application specific, has been left for implementation at a higher level. This allows the use of general purpose processors which can be programmed to express the complexity of these functions using high level programming languages.

Our analysis of the algorithm has concentrated on investigating the data dependencies in the algorithm allowing us to design regular (systolic) arrays for these operators which exhibits many generic features. The number of cells is independent of the length of the chromosomes and the array is scalable according to population size. Modular designs for GA processing result from appropriate use of migration schemes where the population of each "island" can be chosen to match the device size.

Traditionally regular array design has been aimed at VLSI implementation. Our design is modular and can effectively exploit the emerging technology of Field Programmable Gate Arrays (FPGAs). In particular the aspects of dynamic reconfiguration and flexibility of means that a FPGA implementation of our generic array could form a viable basis for an evolvable hardware system for realising a wide range of optimization algorithms.

We have simulated the macro-pipeline using the occam programming language and confirmed the operation of the genetic algorithm. We are in the process of prototyping the pipeline for implementation onto FPGA. As part of this process we have already designed and implemented the mutation array, which includes the systolic random number generator. It is our belief that the cells in this array will require the greatest number of clock cycles to

execute and so will dictate the overall execution speed of the pipeline. The cell design executed in 9 clock cycles albeit using a basic generator where $\lambda = 3$ and $\mu = 5$. Experimental results derived from simulating the mutation array on a Xilinix XC4006PC84-4 indicated a clock speed of 12.5 MHz. This corresponds to 720ns per mutation or 1.38 million mutations per second. To put this figure into perspective, a population of 100 chromosomes, with a length of 100 genes can be processed by this array in under 8 ms.

References

[1] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[2] Ryohei Nakano. Conventional genetic algorithms for job shop problems. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 474–479. San Mateo: Morgan Kaufmann, 1991.

[3] Hsiao-Lan Fang, Peter Ross, and David Corne. A promising Genetic Algorithm approach to job-shop scheduling, rescheduling and open-shop scheduling problems. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 375 382. San Mateo: Morgan Kaufmann, 1993.

[4] D. C. Mattfeld, H. Kopfer, and C. Bierwirth. Control of parallel population dynamics by social-like behavior of GA-individuals. In Y. Davidor, H-P. Schwefel, and R. Manner, editors, *Parallel Problem Solving from Nature – PPSN III*, number 866 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[5] J.E. Beasley and P. Chu. A genetic algorithm for the set covering problem. *European Journal of Operations Research*, To Appear.

[6] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, pages 2 9. San Mateo: Morgan Kaufmann, 1989.

[7] D. Beasley, D. R. Bull, and R. R. Martin. An overview of genetic algorithms : Part 2, research topics. *University Computing*, (15(4)):170–181, 1993.

[8] J. M. Ahuactzin, Talbi, P. Bessiere, and E. Mazer. Using genetic algorithms for robot motion planning. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 671 675, Austria, Aug. 1992. J. Wiley and Sons.

[9] H. Chan and P. Mazumder. A systolic architecture for high speed hypergraph partitioning using genetic algorithms. In *Lecture Notes in Computer Science*, number 956, pages 109 126. Springer-Verlag, 1995.

[10] S. D. Scott. HGA: A hardware genetic algorithm. Master's thesis, University of Nebraska, Lincon, Nebraska, 1994.

[11] P. Graham and B. Nelson. A hardware genetic algorithm for the travelling salesman problem on SPLASH 2. In *Lecture Notes in Computer Science*, number 957, pages 352 361. Springer-Verlag, 1995.

[12] B. C. H. Turton and T. Arslan. A parallel genetic VLSI architecture for combinatorial real-time applications - disc scheduling. In *Proc. first IEE/IEEE Int. Conf. Genetic Algorithms in Engineering Systems : Innovations and Applications*, pages 493–498, Sept. 1995.

[13] B. C. H. Turton and T. Arslan. An architecture for enhancing image processing via parallel genetic algorithms and data compression. In *Proc. first IEE/IEEE Int. Conf. Genetic Algorithms in Engineering Systems : Innovations and Applications*, pages 337–342, Sept. 1995.

[14] M. Salami and G. Cain. Multiple genetic algorithm processor for the economic power dispatch problem. In *Proc. first IEE/IEEE Int. Conf. Genetic Algorithms in Engineering Systems : Innovations and Applications*, pages 188 193, Sep. 1994.

[15] M. Salami and G. Cain. An adaptive PID controller based on genetic algorithm processor. In *Proc. first IEE/IEEE Int. Conf. Genetic Algorithms in Engineering Systems : Innovations and Applications*, pages 88–93, Sep. 1994.

[16] G. M. Megson. *An Introduction to Systolic Algorithm Design*. Clarendon Press, Oxford, 1992.

[17] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, Vol. 71:113–120, Jan. 1983.

[18] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Proc. Sym. Sparse Matrix Computations and their Applications*, pages 256–282. Society for Industrial and Applied Mathematics, 1978.

[19] C. E. Leiserson. *Area Efficent VLSI computation*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, October 1981.

[20] E. Cantú-Paz. A summary of research on parallel genetic algorithms. Technical Report 95007, IlliGAL, University of Illinois, Jun. 1995.

[21] A. Darte. Regular partitioning for synthesizing fixed-size systolic array. *J. of VLSI Integation*, Vol. 12:293–304, 1991.

[22] D. E. Goldberg, H. Kargupta, J. Horn, and E. Cantú-Paz. Critical deme size for serial and parallel genetic algorithms. Technical Report 95002, IlliGAL, University of Illinois, Jan. 1995.

[23] H. T. Kung and M. S. Lam. Wafer-scale intergration and two-level pipelined implementation of systolic arrays. *Journal of Parallel and Distributed Computing*, pages 32 63, 1984.

[24] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, MA, 1989.

[25] I. M. Bland and G. M. Megson. Systolic random number generation for genetic algorithms. *Electronic Letters*, Vol. 32(12):1069, 1996.

[26] B. Jansson. *Random Number Generators*. Victor Pettersons Bokindustri Aktiebolag, Stockholm, 1966.

[27] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. Rawlings, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, California, 1991.

[28] J.E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 14 21, 1987.

[29] J. Branke, H.C. Andersen, and H. Schmeck. Global selection methods for SIMD computers. In *Proceedings of the AISB96 Workshop on Evolutionary Computing*, To Appear.

[30] R. Tanese. Distributed genetic algorithms. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*. San Mateo: Morgan Kaufmann, 1989.