A Machine-Independent Analysis of Parallel Genetic Algorithms

V. Scott Gordon
Computer and Information Science
Sonoma State University
Internet: gordons@sonoma.edu

Darrell Whitley
Department of Computer Science
Colorado State University
Internet: whitley@cs.colostate.edu

Abstract

This paper presents a *machine-independent* study of parallel genetic algorithm performance. Our approach utilizes a dataflow model of computation in conjunction with Sisal, an implicit parallel programming language. We compare problem-solving power and runtime efficiency for several parallel genetic algorithms under uniform conditions. The proposed method makes it possible to identify all sources of potential parallelism, and to locate and measure bottlenecks. The dataflow model thus provides a systematic way to develop and evaluate genetic algorithms.

1 Introduction

This paper presents a machine-independent study of parallel genetic algorithm performance. Genetic algorithms are search algorithms loosely based on principles of natural evolution, particularly genetic evolution. Researchers became widely interested in parallel implementations of genetic algorithms when it became apparent that the parallel models were often outperforming the serial models, even when the parallel models were executed serially. By using a machine-independent analysis and thereby avoiding the more typical machine-driven *ad hoc* approaches, exploitable parallelism is revealed, in some cases where it was not previously known to exist.

In order for any comparative study of parallel genetic algorithms to be useful, two factors must be considered: problem-solving power and execution efficiency. It is reasonable to assume that structural changes in a genetic algorithm designed to enhance parallelism will impact its ability to solve problems. If the changes result in slower problem-solving behavior, then it might be necessary to determine how much parallelism would have to be achieved in order to compensate for this loss. Conversely, if the changes result in faster problem-solving, then the parallel models might be preferable even in the absence of parallel hardware. In this paper these issues are examined by analyzing the potential parallelism of a wide variety of parallel genetic algorithms and by testing the problem solving abilities of these algorithms on a suite of function optimization tasks.

The genetic algorithms are coded in Sisal, an implicit parallel programming language, and executed on a simulator of the Manchester Dataflow Machine [GKB87]. We compare

their execution characteristics by analyzing the various machine-independent measures generated by the simulator such as critical path length and average parallelism. The simulator also provides ideal parallelism profiles of the various algorithms. This approach is not without limitations. The dataflow simulator reports *all* sources of parallelism, some of which are not realizable in a practical sense. The Sisal code can be altered, however, to mask parallelism that is judged to be implausible. Further, we have found that by introducing pseudo-dependencies at various spots in the genetic algorithm, it is possible to use the critical path information output by the simulator to measure the severity of serial bottlenecks. Finally, each algorithm is run on a variety of optimization problems, and the results are normalized by the number of function evaluations.

2 Parallel Genetic Algorithms

A variety of schemes for parallelizing genetic algorithms have appeared in the literature. Some of the parallel implementations are very different from the "traditional" genetic algorithm proposed by Holland [Hol75], especially with regards to population structure and selection mechanisms. To date, implementing parallel genetic algorithms has involved using either a single population using global random mating and replacement, or a population with some form of locality imposed on mating and replacement strategies. Researchers have recognized that this same issue is at the heart of the debate between Wright and Fisher concerning the role of spatial structure in natural genetic populations (e.g., Mühlenbein [Müh91a], Collins [CJ91]). Wright claimed that modeling biological evolution requires considering local interaction and spatial isolation [Wri32]. These views were challenged by Fisher, who believed that biological evolution could be modeled with unrestricted interaction between individuals [Fis58]. These processes are called *local* and *panmictic* interaction, respectively, and each has been used as a biological analogy for various genetic algorithms.

Parallel genetic algorithm implementations can be separated into three categories: *global*, *island*, and *cellular*. *Global models* use a single population and unrestricted random selection and mating. This includes variants of Holland's canonical genetic algorithm and Goldberg's SGA [Gol89a], as well as Whitley's Genitor [WK88] and Eshelman's CHC [Esh90]. *Island models* achieve parallelism by replicating the approach used in global single population genetic algorithms across subpopulations. Subpopulations communicate by the occasional migration of strings. The general island model allows migration to occur between all subpopulations; in the variant known as the "stepping stone model" the subpopulations have a specific physical arrangement and migration only occurs between subpopulations that are locally near to one another [GrS91]. *Cellular models*, sometimes called "massively parallel" or "fine grain" genetic algorithms, assign one individual per processor and limit mating to a set of nearby strings. Each individual is processed in parallel at each generation.

When implementing traditional genetic algorithms, global information is sometimes required, such as the *average fitness* of the strings in the population. One way to reduce the need for global information (and thus decrease overhead in certain parallel implementations) is to use *tournament selection* [Gol90]. In one implementation of tournament selection, two strings are randomly sampled, and the string with the higher fitness is passed into the intermediate generation. This process continues until the intermediate generation is filled. We

can expect the best string to have two copies in the intermediate generation, the median string to have one copy, and the worst string to have no copies. Goldberg and Deb [GD91] showed that the resulting selective pressure is identical in expectation to a linear ranking. All of our implementations utilize tournament selection.

2.1 Global Models

Serial genetic algorithms such as SGA [Gol89a], Genitor [WK88] and CHC [Esh90] typically employ a single population without locality considerations. Parallel implementations of global population models have not received much attention, because machine-dependent considerations led implementors toward other approaches. Some methods for parallelizing global models have been suggested by Goldberg [Gol89a].

SGA and **Elitist SGA:** Goldberg's Simple Genetic Algorithm (SGA) [Gol89a] is a well known variation on Holland's original model [Hol75]. In SGA, selection, crossover, and mutation are used to generate successive populations of a constant size.

SGA with tournament selection is highly parallelizable. Although the generation-to-generation processing is done serially, string manipulations within a generation can be done simultaneously. This is because the creation of a new individual does not depend in any way on the creation of another. Every two slots of each new generation are replaced by the offspring of two selected parents from the previous generation. Thus $n/2$ processes are utilized (where n is the population size), each of which generates two members of the next generation. Each process also performs crossover (if necessary), and mutation. In SGA, there is no guarantee that the very best individual in a population will survive. In the *Elitist SGA* a copy of the best individual is always placed in the next generation.

We divide the population into two interleaved halves, denoted *even* and *odd*. For each pair of slots in the population, four individuals are randomly selected from the previous generation. Each slot conducts a tournament, keeping the best of two strings for reproduction. Each process thus generates two offspring, and these pairs are collected to form new odd/even population halves. This is illustrated in Figure 1. *Probability of crossover* is simulated by denoting beforehand which processes perform crossover and which do not. Since strings are randomly assigned locations in the intermediate generation, this is sufficient to achieve a random chance of undergoing crossover for arbitrary string pairs according to the crossover probability. In Figure 1 crossover does not occur below the heavy line.

pCHC: CHC is a genetic algorithm model developed by Eshelman [Esh90]. CHC is similar to SGA except for the following: (1) the best n strings are extracted from both generation t and generation $t+1$ to form the new generation; (2) parents are paired through incest prevention (i.e., by selecting parents which are distant from one another in Hamming space), rather than according to fitness; (3) mutation is not performed until the population has converged to within some convergence criterion, and then is done at a very high rate (typically 35%) to reinitialize the population; and (4) *HUX* (heterogeneous uniform crossover) recombination [Esh90] is used. CHC uses *survival* of the fittest rather than *selection* of the fittest, because fitness does not play a role in selection. Rather, fitness is used to determine which offspring survive into the next generation.

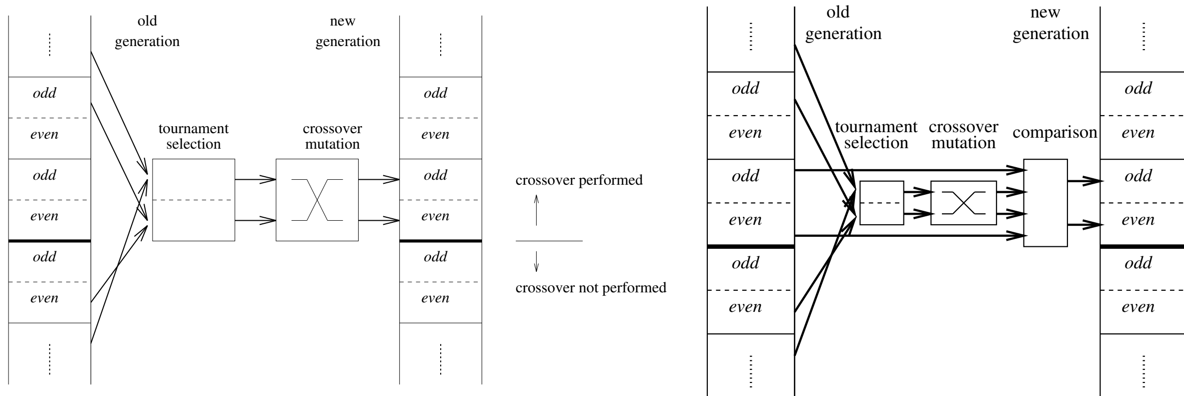


Figure 1: Parallel-SGA (left) and pCHC (right). The vertical columns represent populations before and after a single generation.

Although CHC has been fully implemented in Sisal, it is not as well suited for parallel execution as SGA. This is because selecting the best n from $2n$ strings involves a global operation equivalent to sorting. An illustration of a more parallel algorithm, hereafter called *pCHC*, which simulates CHC, is shown in Figure 1 (also described in [GWB92]).

In pCHC, each process compares its two newly-produced offspring in generation $t+1$ against two particular elements from generation t, (based on iteration number), and retains the best two of the four. This approach results in an algorithm similar to CHC but with a weaker selective pressure. Note that pCHC does not guarantee that the best n out of the $2n$ individuals will survive, but it does guarantee that at least the best two individuals will survive. Tournament selection is used not to select the fittest strings, but to select pairs of individuals which are relatively dissimilar (in Hamming distance) for mating.

Genitor: Normally in the Genitor algorithm [WK88] rank-based selection is applied to a sorted population. Two parents are selected and a single offspring is produced that displaces an individual with low fitness. Our current implementation utilizes tournament selection and replacement to eliminate the need to keep the population in sorted order. The winners of a small tournament recombine and the offspring replaces the loser of the tournament if the offspring has a higher fitness. Whereas in other implementations of Genitor one of the two possible offsprings is chosen randomly before evaluation, here both offspring are evaluated and the best one is retained.

2.2 Island Models

Island models were developed for use on machines with a relatively small number of processors. In a parallel island model, the population is divided into smaller subpopulations, each of which reside on a separate processor. Periodic migration of small numbers of individuals between processors allows locally-converged solutions to mix. Many researchers have reported excellent results with this approach. For example, Starkweather et. al. have shown

improved empirical results on several problems over single population genetic algorithms even when the total number of strings and functions evaluations are the same [SWM91]. Tanese has proposed that the island model is further enhanced when each subpopulation utilizes different parameter settings [Tan89].

Island SGA and **Elitist Island-SGA:** Each "island" is an SGA with its own subpopulation. Migration between islands uses a *ring* topology, and a single individual is chosen for migration by tournament selection, where the losing individual is replaced by the winning individual from the adjacent subpopulation. The Elitist Island-SGA involves a straightforward insertion of the Elitist-SGA model into the Island model (in place of SGA). Unlike Island-SGA, the *best* string in each subpopulation is the only one that migrates, since the use of elitism guarantees that the best string has already been identified.

Island-pCHC and **Island-Genitor:** These are straightforward insertions of pCHC and Genitor into the Island model (in place of SGA). Migration is the same as in Island-SGA. The Island-Genitor model is analogous to Genitor-II [WS90], except that we use a migration tournament as described above rather than migrating the very best strings.

2.3 Cellular Models

Cellular genetic algorithms (CGA) are designed to take advantage of massively parallel machines. In the simplest case one can use a single large population with one string per processor. In order to avoid high communication overhead, CGA's generally impose limitations on mating based on distance. For example, strings may conceptually be arranged on a grid, and only be allowed to perform crossover with nearby strings. This leads to a form of locality known as "isolation-by-distance" [GSc89]. It has been shown that groups of similar individuals (also called *niches*) can form, and that these groups function in ways that may be similar to discrete subpopulations (islands). However, there are no actual boundaries between the groups, and nearby niches can more easily be overtaken by competing niches than in an island model. At the same time, distant niches may be affected more slowly. Davidor calls this model the *ECO GA* [Dav91].

We consider cellular genetic algorithms wherein cells reside on a two-dimensional torus (the individual cells form a square grid where the edges of the grid wrap-around). Each resident individual (i.e., string) selects a mate from nearby strings, and the offspring may replace the resident individual depending on the replacement strategy used. The neighborhood from which a mate is selected is called a *deme*. Each string is processed in parallel at each generation. We consider three approaches: *fixed topology*, *random walk*, and *island*.

In a *fixed topology* implementation, demes consist of the strings residing in particular grid locations near the cell in question. The same locations comprise the deme for every generation. An individual is selected from the set and crossover is performed with the original individual. (In our experiments, we consider various selection schemes). The offspring replaces the original individual depending on their relative fitness. It can be shown that this type of cellular genetic algorithm is a finite cellular automata with probabilistic rewrite rules, where the alphabet of the cellular automata is equal to the set of strings in the search space [Whi93].

In a *Random Walk* implementation, a cell's deme consists of the strings which reside along a random walk starting at that cell. Each string along the walk is compared and the most fit string is chosen for mating. The length of the random walk is a tunable parameter. In our implementations, steps can be up, down, left, or right. It is possible for the walk to traverse the same cell more than once, and may traverse the resident individual.

In an *Island CGA*, the grid is divided into smaller subgrids, each of which is a torus that is processed separately using one of the above methods. Migration between subgrids takes place at predefined intervals.

3 The Dataflow Approach and Sisal

Sisal is a functional side-effect-free language which provides a machine-independent way of studying the implicit parallelism in computer programs. There are several advantages to using Sisal to write parallel programs. First, programs written in Sisal can be evaluated using a dataflow simulator which collects statistics while it executes the code. The dataflow simulator generates parallelism profiles and various machine-independent measures such as critical path length and average parallelism. Second, Sisal programs can be ported without modification to a variety of parallel architectures, including the Cray 2 vector processor, the shared memory Sequent and Alliant multiprocessors, the distributed memory nCUBE 2, as well as the Manchester Dataflow Machine.

In the dataflow model, a program is represented as a dependency graph, and data flows directly between nodes rather than via shared variables. Since each node can perform its task immediately upon receiving all of its required input, parallel execution can occur at an arbitrarily fine-grained level. The dataflow model is thus capable of expressing all forms of parallelism, which makes it an ideal framework for analyzing parallel algorithms. An algorithm's inherent parallelism and sequential threads are exposed when implemented in a dataflow environment [Böh90].

Some of the fine-grained parallelism exposed by using Sisal is exploitable on SIMD architectures. For example, the Cray-2 Sisal compiler should vectorize many of the parallel features described in this paper (where noted). Much of the courser-grained parallelism exposed by using Sisal is exploitable on MIMD architectures, some of which has been described by other researchers [Müh91b].

Parallel profile and critical path information also allows one to examine the sensitivity of various algorithms to parameter changes (such as population size, string length, etc.) from a machine-independent point of view. This information may help us develop more efficient parallel genetic algorithms depending on parametric constraints.

3.1 Average Parallelism and Critical Path

When a Sisal program is executed on the dataflow simulator, simulation proceeds in discrete *time steps* during which all enabled instructions (those for which data is available) are executed. It is assumed that every instruction executes and sends its resulting data to its successor instructions in exactly one time step. The lack of side-effects in Sisal allows for highly optimizable compilation into parallel code, thus exposing inherent parallelism which

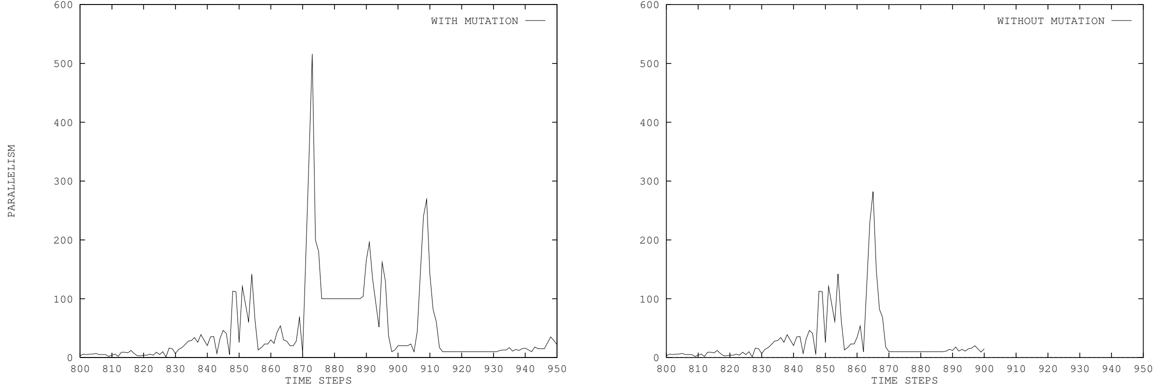


Figure 2: Profiles for SGA with mutation (left) and without mutation (right). Here we use a trivial evaluation function so that the profiles primarily illustrate genetic operators acting on the population. For this example, population size and string length are both 10.

is reported after execution.

Two time-related measurements are recorded: S_1, the total number of instructions executed, and S_∞, the total number of time steps required (also called the *critical path*). *Average parallelism* is defined as S_1/S_∞. The simulator reports values for average parallelism, S_1, and S_∞ at the conclusion of program execution. These figures are for runs which include initialization (i.e., randomly generating the population, etc.). More interesting measures are the average parallelism and critical path for a single generation, which can be obtained by running each algorithm for 3 generations and then for 2 generations, and calculating:

$$\frac{S_1}{S_\infty} = \frac{S_1^3 - S_1^2}{S_\infty^3 - S_\infty^2} \qquad\qquad S_\infty = S_\infty^3 - S_\infty^2 \tag{1}$$

The parallelism of the program can be displayed in a *parallelism profile* by plotting the number of executed instructions at each time step. For example, Figure 2 shows two parallelism profiles of a single generation of SGA, the first with mutation turned on and the second with mutation turned off. The X-axis is the number of time steps along the critical path, and the Y-axis is the number of executed instructions at each time step. The higher the value, the greater the parallelism.

The profiles in Figure 2 suggest the mutation operator used is highly parallelizable, as indicated by the very tall "spike" in the graph on the left. Parallelism profiles like this allow us to isolate key operators in terms of their impact on parallelism. Examination of the Sisal code alongside the parallelism profile also helps reveal where additional parallelism in the algorithms might be located, since low, flat areas indicate the absence of parallelism.

Parallelism profiles are highly problem-dependent. The profiles shown in Figure 2 utilize a trivial evaluation function (one that simply returns a constant value in a single time step) so that the profiles represent just genetic operations. Figure 3 shows profiles for one generation of SGA with the trivial evaluation versus an actual optimization problem (DeJong's F2). Clearly the critical path is heavily affected by the function (and, in this case, the Graycode translation which precedes it). For more complex functions, one might expect the evaluation function to dominate the profile.

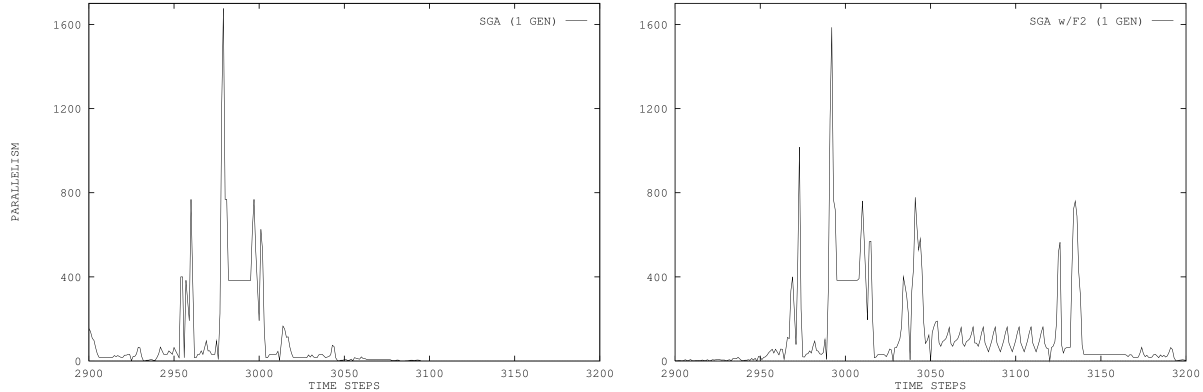


Figure 3: Profiles for SGA with trivial function (left) and F2 (right). F2 involves Graycode translation and a simple computation. Population size = 16, string length = 24.

3.2 Distinguishing *Exploitable* Parallelism

In some cases, the parallelism profiles express parallelism that is unrealizable because the dataflow simulator makes a number of assumptions regarding machine characteristics. First, the simulator assumes that each instruction can be executed in one time step. This is unrealistic when the population is distributed across several processors, because of communication overhead. The extent to which communication overhead is a problem depends on how well the implementation utilizes locality. Second, the simulator assumes that an unlimited number of processors are available. Generally this is not the case. Even if unlimited processors were available, the simulator does not take into account overhead associated with processor allocation or communication. Third, it is assumed that parallelism can be utilized at an arbitrarily fine grained level. However, one cannot expect to have such an ideal environment. On a real machine, some of the genetic operators would have to be implemented at a courser grain. For example, in the previous simulations a mutation operator was used that caused every bit to be mutated in parallel. Since the dataflow simulator assumes an unlimited number of processors with no usage overhead, it reports that mutation of the entire population can occur in exactly the same time as it takes to mutate a single bit. While this is theoretically valid, a real machine would incur substantial overhead to allocate the processors and collect the results. Thus on a machine without parallel bit operations, one would prefer a different mutation operator even if it caused the critical path to grow with increases in population size and/or string length.

Parallelism profiles do, however, contain useful information when analyzed in a systematic manner. The profiles can be viewed as an expression of the maximum parallelism, and therefore should be adjusted to eliminate unrealistic parallelism. One approach is to examine the code alongside the profile in a subjective manner. Section 4 describes a more rigorous method whereby the Sisal code is changed so as to mask parallelism that seems implausible.

3.3 Parallelism Profiles of the Algorithms

Table 1 shows average parallelism and critical path for *one generation* of two global models (SGA, pCHC), an Island model (Island-SGA), and a fixed topology cellular genetic algo-

rithm. Data is shown both for population sizes of 16 and 100. The values for the Island model are for a combination of one generation and one migration.

Figure 4 contains parallelism profiles of three generations for SGA, pCHC, Island, and cellular genetic algorithms, similar to those reported earlier by Gordon *et al.* [GWB92]. Note that in the Island model, migration occurs between the second and third generations. Profiles for Genitor will be examined later in the paper.

Algorithm	PopSize	Avg parallelism	Critical path	Instr executed
SGA	100	658	137	81227
SGA	16	109	137	14996
pCHC	100	520	185	96353
pCHC	16	86	184	15754
Island-SGA	(20x5) 100	351	275	96731
Island-SGA	(4x4) 16	66	275	18326
Cellular	100	1061	155	164510
Cellular	16	171	155	26505

Table 1: Average parallelism and critical path measures

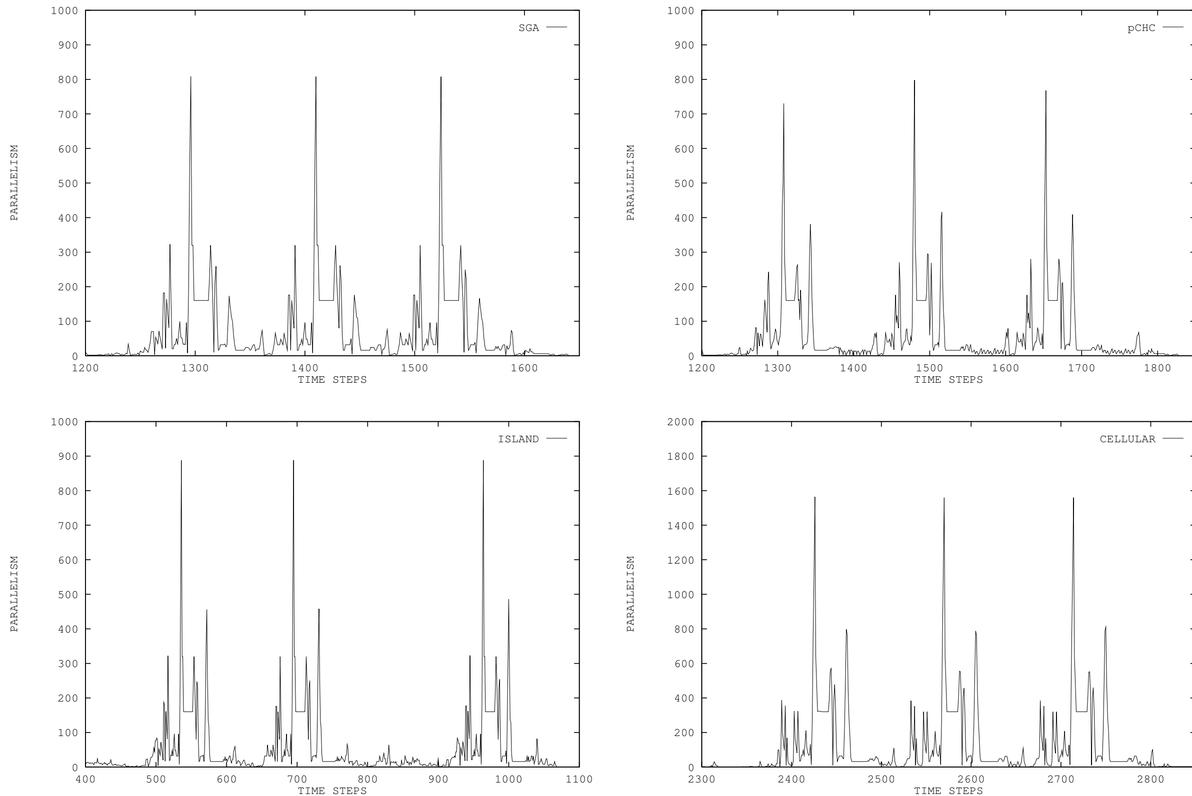


Figure 4: Parallelism profiles for generational models (SGA, pCHC, Island, and Cellular). Population size is 16 and string length is 10. Three generations are shown.

Clearly, for every model, the critical path length is independent of population size. This means that in an ideal environment (access to an unlimited pool of processors with no usage overhead) increasing the population size does not increase the critical path length. Since one of the goals of parallel genetic algorithms is facilitating larger population sizes, this is an encouraging result.

The Island Model has the longest critical path length in part due to the inclusion of migration into the statistics. The data indicates that the cellular algorithm performs more operations and has higher average parallelism than SGA, but is not faster than SGA since its critical path length is slightly longer.

It is remarkable that the profiles look so similar for all of of the models. This suggests that while implementors may be doing a good job of mapping the algorithms to architectures, very little has been done to change the nature of the underlying parallelism inherent in the genetic algorithm. In actuality, *researchers have introduced locality into the algorithms so that the inherent parallelism can be exploited*.

3.4 Operator Sensitivity to Parameters

The dataflow simulator makes it possible to isolate the impact of genetic operators on parallelism [GWB92] The parallelism profiles provide a way of measuring the sensitivity of each operator's parallelism to different parameter settings. For example, consider two different mutation operators *bitwise* and *next-point*. Bitwise mutation is a fully parallel loop in which each bit in the population is tested individually for possible mutation. Next-point mutation attempts to correct the inefficiencies of bitwise mutation by calculating the next bit to mutate as follows:

$$NextPoint = \lfloor \frac{Rnd(0,1)}{probm} \rfloor + CurrentPoint + 1 \quad (2)$$

where $probm$ is the probability of mutation per bit. This is applied until $NextPoint > strlen$, where $strlen$ is the string length.

Table 2 shows average parallelism and critical path for one generation of SGA for both mutation operators, with population sizes of 100 and 16, and equal mutation rates. Note that both are insensitive to population size (critical path is unaffected by changes in population size). Bitwise mutation at first appears more efficient than next-point mutation due to the higher average parallelism and the shorter critical path. However, it is unlikely that the parallelism in bitwise mutation is *exploitable*, because it requires parallel bit operations.

Algorithm	PopSize	Avg parallelism	Critical path	Instr executed
SGA/bit	100	658	137	81227
SGA/bit	16	109	137	14996
SGA/nxtpt	100	282	271	76507
SGA/nxtpt	16	45	271	12087

Table 2: Mutation operators; sensitivity to population size

4 Using the Dataflow Model to Measure Bottlenecks

By introducing pseudo-dependencies at various spots in the Sisal implementation of a genetic algorithm, it is possible to mask parallelism at certain levels in order to expose parallelism at other levels. This helps in locating parallelism bottlenecks in the algorithms.

For each genetic algorithm, all of the loops in the code are identified, then categorized as *bit-level*, *string-level*, or *population-level*, depending on whether they iterate through an individual, several individuals, or several populations, respectively. A loop which depends on the string length is bit-level, a loop which depends on the population size is string-level, and a loop which depends on the number of subpopulations is population-level. For each loop, the change in critical path over changes in magnitude of the associated parameter is measured. When the loop cannot be executed in parallel, the magnitude of the change represents a potential bottleneck. When a loop *can* be executed in parallel, a bottleneck *still may exist*, because of overhead factors such as those described earlier. In this case the potential bottleneck can be measured by modifying the code to force serial execution. This is done by introducing an artificial dependency, typically by adding a dummy variable which is referenced at the start of the loop and modified at the end of the loop.

A small amount of residual parallelism remains after serializing a loop, because the simulator identifies all instruction-level parallelism. The measured values and the actual values differ by a constant, and it is assumed that the residual instruction-level parallelism is sufficiently small that it could be exploited on most modern architectures. Thus it is argued that the reported critical path values, while not representing absolute measures, do constitute a valid relative ordering with respect to their sensitivity to changes in magnitude of the relevant parameter.

Identical bit-level bottlenecks occur in several of the algorithms, so they are measured in SGA (except for one which is specific to pCHC). String-level bottlenecks vary among the algorithms, and are measured for SGA, Genitor, and the Cellular algorithm. Population-level bottlenecks are applicable only to Island models, so they are measured in Island-SGA.

Finally, for each bottleneck, the practical implications of the measurements are discussed. Some of the bottlenecks are easier to remove than others. For example, some of the bit-level bottlenecks may be improvable via vectorization, while or others may be difficult or impossible to correct. The seriousness of some bottlenecks depends on the application.

Bottlenecks associated with particular optimization problems are not considered. Rather, the measurements concentrate on the genetic operations themselves. However, for some problems, the overhead for the fitness function can have more influence on performance than the performance of the genetic operators. Note that when there is a high-degree of exploitable string-level parallelism, overhead associated with a particular optimization problem is incurred only once per generation, since each of the strings can be evaluated in parallel.

4.1 Bit-Level Bottlenecks

Bit-level bottlenecks appear by far most often in loops which depend on *string length*. Less often, a loop depends on the *crossover rate* or the *mutation rate*. Table 3 lists the bit-level bottlenecks and their associated severity depending on the value of *string length*. This information is also shown graphically in Figure 5. The values were obtained by using SGA

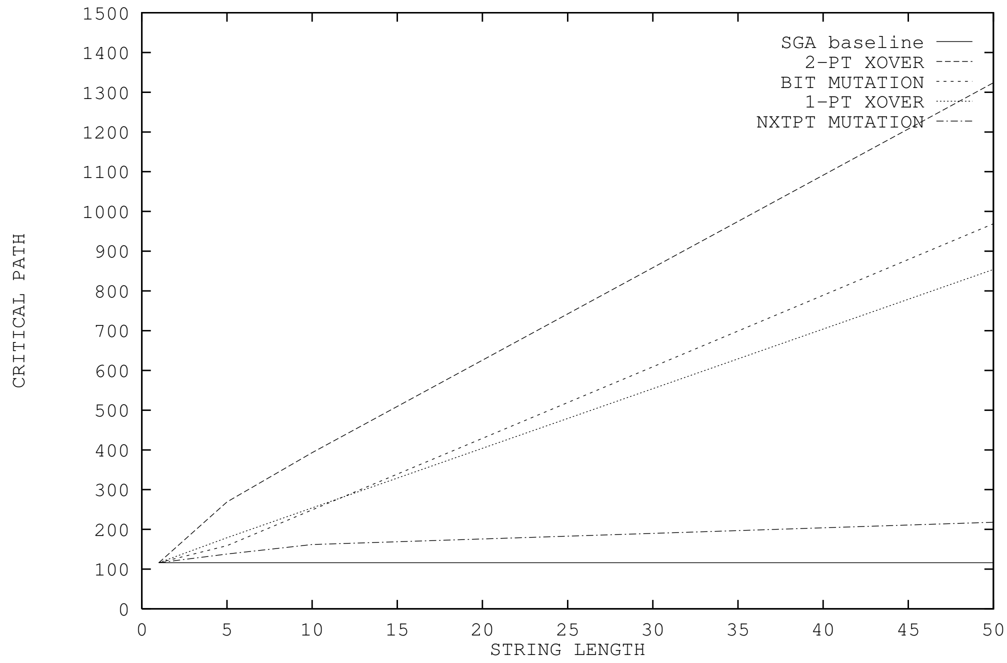
(except where noted) with bitwise mutation and 1-point crossover. A serialized version of the various operators was used, replacing the bitwise mutation or 1-point crossover when appropriate. Thus, variance in bottleneck severity is due only to the specific operators indicated. Critical path information for the fully parallelized version is shown in the rightmost column, and represents an upper bound on the performance. Some of the loops (such as for nextpoint mutation) cannot be parallelized because of dependencies.

Critical path length for one generation, various operators.						
genetic operator	serialized execution			parallel execution		
strlen =	5	10	50	5	10	50
Bitwise mutation	159	249	969	116	116	116
Nextpoint mutation	131	162	218	*no change*		
1-pt crossover	179	254	854	116	116	116
2-pt rs crossover	269	393	1324	209	266	409
DeGray	161	197	477	*no change*		
Decode	176	244	787	115	126	206
Hamming Dist (pCHC)	221	276	716	185	195	275

Table 3: Measures of bit-level bottlenecks in genetic algorithms

Decode converts a binary string to an integer value, and *DeGray* converts Gray-coded strings to binary-encoded strings. Both are described by Goldberg [Gol89a] and both represent fairly substantial bottlenecks. Hamming distance calculation is used in the pCHC algorithm to select parents via incest-prevention.

Exploiting the parallelism reported in Table 3 is not always easy. For example, it is unlikely that the parallelism achieved for bitwise mutation is exploitable, and the operation does not lend itself well to vectorization. But the bottleneck is far less severe for next-point mutation. It is possible that the parallelism in one-point crossover *can* be exploited by vectorization if appropriate masks can be randomly generated at run-time. The parallelism in Decode and Hamming distance calculation should also be exploitable by vectorization. The bottleneck shown for two-point crossover is more severe, and the dataflow simulator is unable to report much inherent parallelism. Further, two-point crossover may be difficult to vectorize because the masks that would have to be generated are sufficiently complex that the effort creating them may be as great as the effort to perform crossover. DeGray is also not easily parallelized, and does not lend itself to vectorization, since there are dependencies across the bitstring.



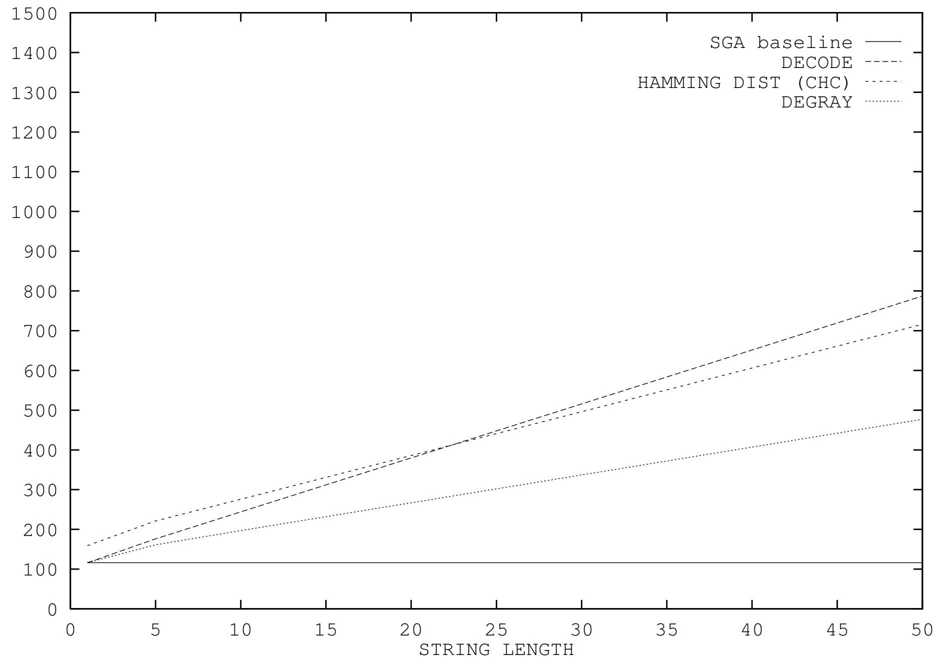


Figure 5: String-Length vs. Critical Path for various operators. Operator sensitivity to changes in string-length is shown, along with the upper-bound on parallelism (shown here as *SGA baseline*).

4.2 String-Level Bottlenecks

String-level bottlenecks in genetic algorithms occur in loops which depend on *population size*. Table 4 lists the string-level bottlenecks and their associated severity depending on the value of *population size*. The measurements for SGA and the cellular genetic algorithm are for one generation, and were obtained by executing a serialized version of the relevant loop which processes each string. Critical path information for the fully parallelized version is shown in the rightmost column, and represents an upper-bound on the performance. The same information is shown graphically in Figure 6.

Critical path length – one SGA generation.				
	severity		corrected	
popsize =	16	100	16	100
SGA	900	5782	116	116
Cellular	1218	7573	134	134
Genitor	1084	6858	*no change*	

Table 4: Measures of string-level bottlenecks in genetic algorithms

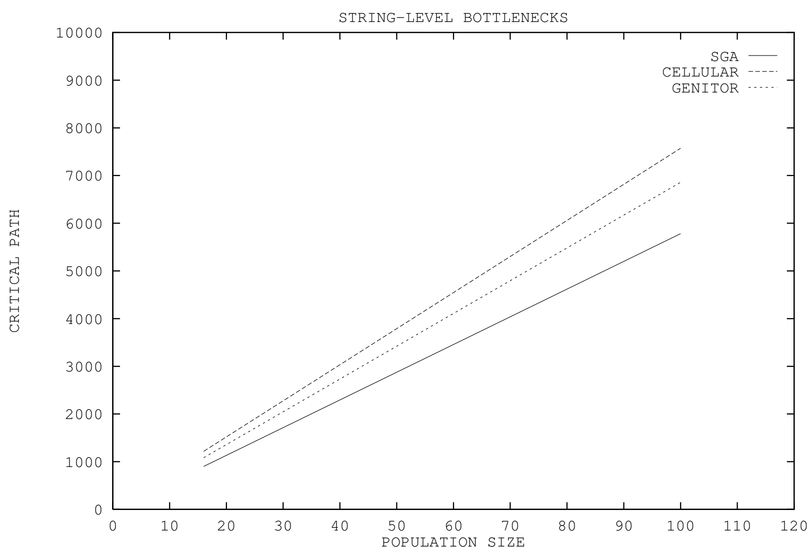


Figure 6: Population size vs. Critical Path for various algorithms. The magnitudes of the serial bottlenecks of SGA, Genitor, and the cellular model are shown.

In order for measurement comparisons to be meaningful across the three genetic algorithms, normalization is done so that the amount of work expressed per time unit is comparable. We use *number of function evaluations*, and critical path lengths are expressed in terms of one *SGA generation*. In Genitor, strings are necessarily processed serially, so the bottleneck can be measured directly. The results for Genitor are normalized as follows: Genitor performs two function evaluations per generation but only one string is retained. Thus it requires $n/2$ (where n is the population size) generations in Genitor to perform the same number of function evaluations as one SGA generation. Therefore when comparing Genitor's performance with one generation of SGA, Genitor is executed for $n/2$ generations.

A similar normalization is done for the cellular genetic algorithm. This implementation of the cellular genetic algorithm performs two function evaluations for each location in the 2-D grid. Thus in one generation it performs twice the number of function evaluations as SGA does. Therefore when comparing the performance of the cellular genetic algorithm against SGA, the results are divided by two. These normalizations have been described previously by Gordon *et al.* [GW93a].

The dataflow simulator indicates that SGA can be fully parallelized at the string level by using $n/2$ processes (where n is the population size), each of which generates two members of the population. Each process must have access to the entire population because of the nature of selection in SGA, necessitating the use of a shared memory system. Since tournament selection is used, the $n/2$ processes will access the same population structure $tn/2$ times (where t is the tournament size). This would likely result in a heavy communication bottleneck [Gor94a].

The cellular genetic algorithm can be fully parallelized at the string level by using n processes (where n is the population size), each of which generates one member of the population. Each process needs access only to those individuals in its deme. Since deme size is fixed, communication overhead remains constant regardless of the population size. Communication overhead in this case is not a bottleneck, and the parallelism is exploitable, most obviously on a grid of processors.

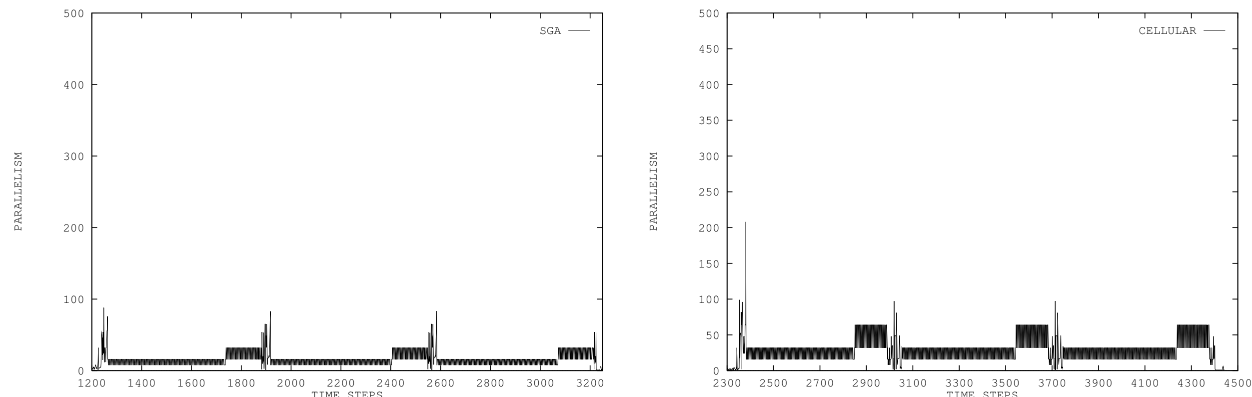


Figure 7: Profiles for SGA (left) and the cellular GA (right), with bit-level parallelism masked.

Figure 7 contains parallelism profiles of SGA and the cellular genetic algorithm, with the bit-level parallelism masked (population size is 16). Compared with the graphs in Figure 4, these profiles are flatter and more elongated, reflecting the removal of bit-level parallelism. The first long, flat area represents crossover and mutation, and the second, somewhat higher flat area represents decode and evaluation of the two child strings. These only appear once per generation, since both algorithms are fully parallel at the string level.

Genitor: The simulator does not report any string-level parallelism for Genitor because of the data dependency associated with inserting one offspring at a time into the population. However, it is possible to execute Genitor with a fairly high degree of parallelism by reordering the operations. Rather than generating the tournament and then immediately performing recombination, we can instead generate triples of indices in the range (1,*popsize*) *until a collision occurs* (that is, until we generate an index which was generated before), and

then perform the recombinations in parallel. The larger the population size, the less chance of collision, and the greater the resulting parallelism. Figure 8 shows parallelism profiles both for Genitor and the revised version of Genitor, assuming that two tournaments are generated before a collision occurs. In this example, we have run 10 string replacements, with a population size of 10 and a string length of 10.

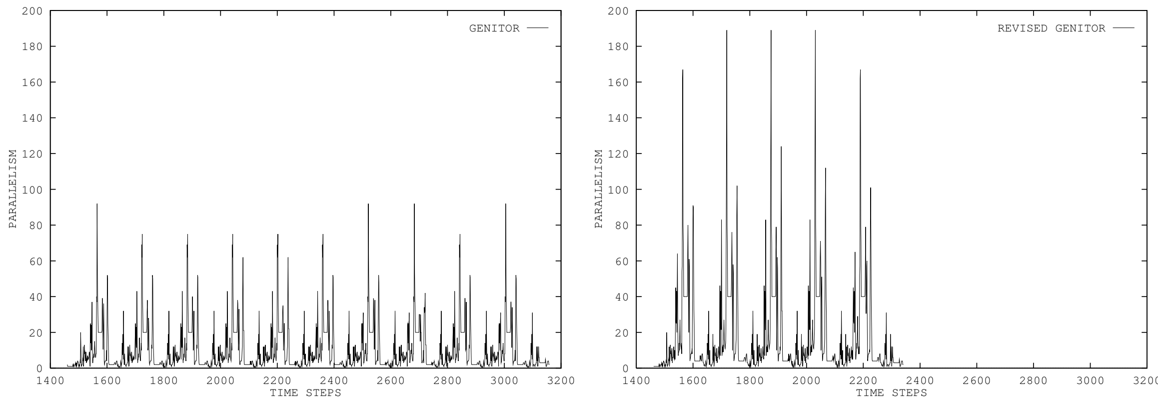


Figure 8: Standard Genitor (left) and revised (right) – parallel profiles. In the profile on the right, we assume that collisions occur after exactly two recombinations.

Note that parallelism is increased roughly by a factor of 2 (the "spikes" are twice as high) and that critical path has been cut in half (the graph is half as wide). Speedup approaches linear as the complexity of the evaluation function increases, because lengthy evaluation functions will dominate the critical path for string generation. Gordon referred to the revised Genitor as "PreGenitor", and showed that in an actual implementation on a shared memory Sequent Balance, speedup was nearly linear even for a simple evaluation function [Gor94b].

The effectiveness of PreGenitor as a parallel algorithm depends on the probability of collisions. It can be shown that the probability of a collision is

$$C_k = 1 - \frac{p!}{(p-k)! \cdot p^k} \tag{3}$$

where p is the population size and k is the number of processors. Gordon [Gor94b] showed how equation 3 can be used to predict the number of processors that can be utilized simultaneously, while still retaining nearly linear speedup. For example, a population size of 100 can utilize 4 processors, and a population size of 500 can utilize 9 processors.

4.3 Population-Level Bottlenecks

Population-level bottlenecks occur in loops which depend on *number of subpopulations*. Table 5 lists the population-level bottlenecks and their associated severity depending on the value of *number of subpopulations*. Since all of the Island models share these loops, measurements are reported just for Island-SGA. Critical path information for the fully parallelized version is shown in the rightmost column, and represents an upper bound on the performance.

Critical path length – one I-SGA generation.				
	severity		corrected	
$\#subpops =$	4	8	4	8
Generate for all subpops	2089	4203	297	297
Migrate between subpops	458	975	110	115

Table 5: Measures of population-level bottlenecks in genetic algorithms

The dataflow simulator confirms that I-SGA can be fully parallelized at the population level by using s processes, where s is the number of subpopulations. Generation-to-generation processing can be done for all subpopulations in a fully parallel fashion because there are no data dependencies and no communication requirements. Migration is also highly parallel, since the determination of which strings migrate can be done for each subpopulation independently, and the migration itself involves little communication overhead. Further, migration only occurs at periodic intervals.

Parallelism exists to varying degrees among all of the genetic algorithms. At the lowest level, parallelism is sometimes exploitable by vectorization. String-level parallelism is equivalent in all of the algorithms except Genitor, which has the least. However, the parallelism that Genitor does contain is fully exploitable, whereas this is not the case for SGA. Genitor is a good choice for shared memory machines with a relatively small number of processors. Island models are preferable for distributed memory machines with a relatively small number of processors. Cellular models are best suited for massively parallel machines because they are best able to exploit string-level parallelism.

5 Problem-Solving Performance

To examine problem-solving power, several parallel genetic algorithms are compared across a wide range of optimization functions. Serial execution is assumed in order to measure problem-solving power independent of actual parallel execution. The genetic algorithms have purposely been reduced to simple configurations in order to compare the effectiveness of basic structural alternatives and selection mechanisms. While it is infeasible to test every possible combination of models, problems, and parameters, it is possible to look for trends.

5.1 Experimental Method

Each of the genetic algorithms described earlier is run on a variety of optimization problems under uniform conditions, and the results are normalized by the number of function evaluations (described earlier in Section 4.2). Two sets of experiments are described, one which compares different genetic algorithms, and one which focuses on cellular algorithms.

No attempt is made to optimize the parameter settings for the various algorithms. Instead, we picked a reasonably general set of parameter values for use throughout the experiments (we did try to adjust mutation for best problem-solving speed). It is well known that

adding local search to a genetic algorithm often improves its performance, but the effects of including local hillclimbing as a supplementary operator are not considered here.

Other researchers have reported faster results [MS93] in optimizing some of the test functions used in our comparative studies. However, different researchers have used different problem representations in their experiments, such as numeric representations, standard binary representation, or gray-coded binary representations. Mathias and Whitley [MW94] have shown that even changing from standard binary coding to a Gray coding can have a dramatic effect on the number of local minima in a test function. Real-valued numeric codings of the parameters will also often induce very different search spaces. This can radically alter the performance of search algorithms, including hillclimbers as well as genetic algorithms. This also creates a dilemma for comparative work. If one's objective is to solve a particular problem, then of course the appropriate thing to do is to pick the algorithm and the associated representation which yields the best performance. However, if one is attempting to compare algorithms and the algorithms are applied to different representations of the problem, then it is unclear that the comparison is a fair way to judge algorithm performance, since the search spaces that are induced by different problem representations are so different. Mathias and Whitley [MW94] have specifically looked at the effect of problem representation for problems in the test suite used here and have shown that representation indeed radically changes the number of local optima and the relative size of the basins of attraction.

For these reasons, we make no comparison outside of the parameter settings presented in this paper. The comparisons used in the paper use a uniform problem representation across experiments for each individual problem in the test set.

In the first set of experiments, nine genetic algorithms are run across the test suite for 30 runs. Population size is fixed at 400 (for island models 8 subpopulations of size 50 are used, and for the cellular model a 20x20 grid is used). Crossover probability is 0.7 (where applicable). Mutation is adjusted to try to maximize problem-solving speed. All of the implementations employ two-point reduced surrogate crossover and next-point mutation. Swap intervals for island models are 5 generations for easy problems and 50 generations for hard ones. In the CGA used here (which we later refer to as *Deme4*), each string recombines with the most fit string among its four nearest neighbors. If an offspring has a higher fitness than the individual, the individual is replaced by the most highly fit offspring.

It is important to note that performance data is reported in two different ways depending on the test problem. Some of the functions are easy for the genetic algorithms and are executed until the global optimum is found in all 30 runs. In this case the average number of generations (and the standard deviation) that it takes for each genetic algorithm to solve the function is reported. For harder problems, the algorithms are run for a set number of generations (normalized to SGA generations). In this case the number of runs (out of 30) in which the global optimum is found is reported, along with the average fitness of the best strings found at the end of each of the 30 runs. Convergence graphs are included for the hardest problems.

In the second set of experiments, we run a variety of cellular genetic algorithms with larger population size ($50x50 = 2500$) on the hardest problems. All of our implementations use two-point reduced surrogate crossover and next-point mutation. Migration intervals for the island models are set to 5 (generations) for easy problems and 50 for hard ones.

5.2 Test Suite

Our test suite includes the following numerical optimization problems: DeJong's original test suite F1-F5 [DeJ75]; Rastrigin, Schwefel, and Griewank functions [MSB91]; Ugly 3 and 4-bit deceptive functions; and Zero-one knapsack problems. These functions were chosen for one or more of the following reasons: (1) they require long bitstrings, (2) they contain many local minima, and/or (3) they appear frequently in genetic algorithm literature.

5.2.1 DeJong Test Suite

DeJong's suite contains five functions. F1 is a linear combination of three identical unimodal nonlinear functions, each with a hamming cliff at the optimum. F2 is a nonlinear combination of two variables, resulting in a "saddle" function. F3 is a linear combination of five identical discontinuous "step ladder" functions. F4 is a linear combination of 30 identical unimodal nonlinear noisy functions, with a large solution space (2^{240}) plus the addition of fairly substantial Gaussian noise. F4 is considered solved when the best string in the population reaches -2.5. F5 is a linear combination of two identical severely multimodal functions (i.e., each contains several local minima). Gray coding was used on F1, F2, and F5.

We carefully considered the pros and cons of including DeJong's functions in our suite because they are sometimes described as being too easy. We decided to include them because they have been widely used by other genetic algorithm researchers, and thus are well understood. Performance results are shown in Table 6. For the island models the swap interval is 5, except F4 where the swap interval is 50.

Generations to solve (*gen*) and standard deviation (*std*) on DeJong's Test Suite for various genetic algorithms										
function	F1		F2		F3		F4		F5	
algorithm	gen	std	gen	std	gen	std	gen	std	gen	std
SGA	30.7	7.4	284	198	16.7	4.2	161	40	14.6	4.4
ESGA	28.9	6.8	83	55	15.3	4.1	153	49	14.3	4.4
pCHC	28.4	6.5	153	139	16.9	3.7	223	104	16.0	3.9
Genitor	17.0	4.1	190	160	8.2	2.1	135	67	7.9	2.5
I-SGA	41.3	11.2	417	253	22.0	5.3	405	192	20.3	6.9
I-ESGA	32.3	7.6	81	40	18.3	5.0	375	197	13.8	4.7
I-pCHC	33.2	7.4	78	57	18.8	4.4	495	239	16.3	5.3
I-Genitor	23.2	5.3	112	94	12.3	3.6	208	162	11.2	3.7
Cellular	32.5	8.0	105	94	17.9	4.6	397	204	15.3	4.3

Table 6: Performance of nine GAs on DeJong's test suite

5.2.2 Rastrigin, Schwefel, and Griewank Functions

The Rastrigin, Schwefel, and Griewank functions have been used by Mühlenbein as a testbed for comparing parallel genetic algorithms [MSB91]. They require longer bitstrings and have more local minima than the functions in DeJong's suite.

Rastrigin's function (F6) is a 200-bit linear combination of 20 tilted sine waves with several local minima. Schwefel's function (F7) is a 100-bit linear combination of 10 functions, each characterized by a second-best minimum which is far away from the global optimum. In the Schwefel function, V is the negative of the global minimum, which is added to the function so as to move the global minimum to zero. The exact value of V depends on system precision; for our experiments $V = 4189.829101$. Griewank's function (F8) is a 100-bit problem similar to Rastrigin, except that the subproblems are multiplied together rather than added, forming a nonlinear combination of the 10 substrings.

Table 7 shows performance data on F6, F7, and F8 after 1000 generations. The migration interval was set to 50 for the island models. Gray coding was used for F6 and F8. Convergence graphs are given in Appendix A.

$$F6\text{:}\quad f(x_i|_{i=1,20}) = 200 + \sum_{i=1}^{20} x_i^2 - 10cos(2\pi x_i), \qquad x_i \in [-5.12, 5.11]$$

$$F7\text{:}\quad f(x_i|_{i=1,10}) = V + \sum_{i=1}^{10} -x_i sin(\sqrt{|x_i|}), \qquad x_i \in [-512, 511]$$

$$F8\text{:}\quad f(x_i|_{i=1,10}) = \sum_{i=1}^{10} \frac{x_i^2}{4000} - \prod_{i=1}^{10} cos(\frac{x_i}{\sqrt{i}}) + 1, \qquad x_i \in [-512, 511]$$

Number of runs solved (*ns*) and average best of 30 runs (*avg*) after 1000 generations.													
function	F6		F7		F8		function	F6		F7		F8	
algorithm	ns	avg	ns	avg	ns	avg	Islands etc.	ns	avg	ns	avg	ns	avg
SGA	0	6.8	0	17.4	0	.161	I-SGA	0	3.8	9	6.5	7	.050
ESGA	2	1.5	16	17.3	1	.107	I-ESGA	13	0.6	13	2.6	3	.066
pCHC	23	0.3	15	5.9	0	.072	I-pCHC	10	0.9	28	0.2	3	.047
Genitor	0	7.9	20	13.2	3	.053	I-Genitor	23	0.2	24	0.9	6	.035
							Cellular	24	0.2	26	0.7	1	.106

Table 7: Performance of nine GAs on Rastrigin (F6), Schwefel (F7), and Griewank (F8).

5.2.3 Deceptive Functions

The ugly 3-bit problem (D3) is a 30-bit artificially-constructed problem introduced by Goldberg, Korb, and Deb [Gol89b] in which ten fully-deceptive 3-bit subproblems are interleaved. In general, the three bits of each subproblem X appear in positions X, 10+X, and 20+X. The ugly 4-bit problem (D4) is a similarly constructed 40-bit problem in which ten fully-deceptive 4-bit subproblems are interleaved [Whi91].

Ugly deceptive problems have very often been misunderstood. These problems isolate interactions in the hyperplane sampling abilities of a genetic algorithm as well as the linkage between bits. This linkage is related to the disruptive effects of crossover. As such, ugly deceptive problems should perhaps be viewed more as analytical tools rather than "hard test problems." Mühlenbein [Müh92] has presented results which indicate such problems can

Ugly 3-bit: Generations to solve (*gen*) and standard dev. (*std*) Ugly 4-bit: # solved (*ns*) and avg best of 30 runs (*avg*) (5000 gens)									
	*D*3		*D*4			*D*3		*D*4	
	gen	std	ns	avg		gen	std	ns	avg
SGA	1547	675	0	22.1	I-SGA	944	273	0	22.2
ESGA	1564	1549	0	9.9	I-ESGA	172	49	30	0.0
pCHC	455	192	17	1.3	I-pCHC	345	139	6	2.0
Genitor	399	133	6	2.1	I-Genitor	439	158	6	2.6
					Cellular	572	240	7	2.4

Table 8: Performance of nine GAs on ugly deceptive functions.

often be solved by hill-climbing; this does not contradict the goals behind the design of fully deceptive problems, since they were not designed to mislead hill-climbing algorithms.

The original ugly deceptive problems [GKD89, WS90] were used to test genetic algorithm implementations that use *no mutation* and a 100% probability of crossover. Whitley *et al.* [WDC92] have shown analytically that using low crossover rates also makes ugly deceptive problems easier to solve (since it reduces the effects of the linkage problem). Also, ugly deceptive problems are particularly designed to mislead traditional simple genetic algorithms of the kind developed by Holland and DeJong and described by Goldberg [Gol89a]. It does not automatically follow that ugly fully deceptive problems are generally hard for other search algorithms, or even for other forms of genetic algorithms.

In the current set of experiments, *all* of the algorithms tested solve the ugly order-3 deceptive problem. SGA, for example, reliably solves the ugly order-3 deceptive problem using a crossover rate of 0.7 and a mutation rate of 1.5%. Further, the Island and CGA models appear to be particularly well suited to solving ugly deceptive functions; different subproblems are solved in various subpopulations (or virtual neighborhoods). The various subproblems can then be exploited and recombined to build up full solutions.

Table 8 shows performance data on the 3-bit and 4-bit deceptive functions (labeled D3 and D4, respectively). The 3-bit problem is easy enough that it is executed until the global optimum is found in all 30 runs. The 4-bit problem, on the other hand, is harder, and each run is executed for a maximum of 5000 generations. It is important to note that performance data is reported differently for the two problems. Migration intervals are set to 5 for the 3-bit problem and 50 for the 4-bit problem in the island models.

5.2.4 Zero-One Knapsack Problems

The zero-one knapsack problem is defined as follows. Given n objects with positive weights W_i and positive profits P_i, and a knapsack capacity M, determine a subset of the objects represented by a bit vector X_i such that:

$$\sum_{i=1}^{n} X_i W_i \leq M \quad and \quad \sum_{i=1}^{n} X_i P_i \quad maximal.$$

A *greedy approximation* to the global optimum can be found by selecting objects by profit/weight ratio until the knapsack cannot be filled any further.

gen and *std* for 20 and 80 object zero-one knapsack problems.									
	$K20$		$K80$			$K20$		$K80$	
	gen	std	gen	std		gen	std	gen	std
SGA	88.7	302.1	32.5	7.9	I-SGA	43.0	31.2	41.0	10.6
ESGA	62.2	64.5	41.9	11.8	I-ESGA	100.3	253.8	33.0	7.7
pCHC	31.3	12.6	32.7	8.2	I-pCHC	34.7	12.9	37.2	8.9
Genitor	24.7	9.9	17.8	4.8	I-Genitor	25.8	17.6	22.0	5.8
					Cellular	32.3	12.5	34.7	7.9

Table 9: Performance of nine GAs on small knapsack problems

A simple encoding scheme is to let each bit represent the inclusion or exclusion of one of the n objects from the knapsack. Thus, a bit string of length n can be used to represent candidate solutions. If the objects are sorted by profit weight ratio, then the greedy approximation appears as a series of "1" bits followed by a series of "0" bits.

The difficulty with this representation is that it is possible to generate infeasible solutions. Setting too many bits to 1 might overflow the capacity of the knapsack. We consider two methods of handling overflow. The first *penalty* method assigns a penalty equal to the amount of overflow. The second method, *partial scan*, adds items to the knapsack one at a time, scanning the bitstring left to right, stopping at the end of the string or when the knapsack overflows, in which case the last item added is removed. The partial scan method has the interesting property that a string of all "1" bits evaluates to the greedy approximation.

The test cases include a 20-object problem and an 80-object problem. The 20-object problem has a global optimum of 445 and a greedy approximation of 275. The 80-object problem has a global optimum of 25729 and a closer greedy approximation of 25713. Gordon, Böhm, and Whitley [GBW94] have argued that genetic algorithms perform poorly on these knapsack problems (and much larger problem instances) when compared with depth-first and branch-and-bound search methods. The knapsack experiments presented here are thus useful only for comparing the various genetic algorithms against each other.

Using our genetic algorithms, the 20-object knapsack problem is *harder* to solve than the 80-object knapsack problem. Further, the *penalty* evaluation method works better on the 20-object problem, and the *partial scan* method works better on the 80-object problem. Performance results are shown in Table 9. The migration interval for island models is 5.

5.3 Summary of Performance Results

Table 10 shows the relative ranking of the genetic algorithms on each of the functions. Table 11 shows the relative performance of the algorithms on each problem, normalized on a scale of 0 to 1 (where 1 is the worst). This is done by dividing each performance value by the worst performance score on that problem. Aggregate values for Tables 10 and 11 are given in the rightmost columns. Table 12 gives the same aggregate information for the hardest problems (F2, F4, Rastrigin, Schwefel, Griewank, D3, D4, and K20), and for the problems with the longest bitstrings (F4, Rastrigin, Schwefel, Griewank, and K80).

Algorithm	F1	F2	F3	F4	F5	F6	F7	F8	D3	D4	K20	K80	Avg	rnk
SGA	5	8	4	3	5	8	9	9	8	8	8	3	6.5	8
ESGA	4	3	3	2	4	6	8	8	9	7	7	9	5.8	7
pCHC	3	6	5	5	7	3	5	6	5	2	3	4	4.5	4
Genitor	1	7	1	1	1	9	7	4	3	4	1	1	3.3	2
I-SGA	9	9	9	8	9	7	6	3	7	9	6	8	7.5	9
I-ESGA	6	2	7	6	3	4	4	5	1	1	9	5	4.4	3
I-pCHC	8	1	8	9	8	5	1	2	2	3	5	7	4.9	5
I-Genitor	2	5	2	4	2	2	3	1	4	6	2	2	2.9	1
Cellular	7	4	6	7	6	1	2	7	6	5	4	6	5.1	6

Table 10: Ranking of performance of nine GAs on test suite

Alg.	F1	F2	F3	F4	F5	F6	F7	F8	D3	D4	K20	K80	Avg	rnk
SGA	.74	.68	.76	.33	.72	.86	1.0	1.0	.99	.99	.88	.78	.81	9
ESGA	.69	.19	.69	.31	.70	.19	.99	.66	1.0	.45	.62	1.0	.62	7
pCHC	.68	.37	.77	.45	.79	.04	.34	.45	.29	.06	.31	.78	.44	3
Genitor	.41	.46	.37	.27	.39	1.0	.76	.33	.26	.09	.25	.42	.42	2
I-SGA	1.0	1.0	1.0	.81	1.0	.48	.37	.31	.60	1.0	.43	.98	.75	8
I-ESGA	.78	.19	.83	.76	.68	.08	.15	.41	.11	0	1.0	.79	.48	6
I-pCHC	.80	.18	.85	1.0	.80	.11	.01	.29	.22	.09	.35	.89	.47	4
I-Genitor	.56	.27	.56	.49	.55	.03	.05	.22	.28	.12	.26	.53	.33	1
Cellular	.79	.25	.81	.80	.75	.03	.04	.65	.37	.11	.32	.83	.48	5

Table 11: Normalized performance of nine GAs on test suite. Normalization is done by dividing each performance value by the worst performance score on that problem.

Algorithm	Avg	rnk	Nrm	rnk
SGA	7.6	9	.84	9
ESGA	6.3	7	.55	7
pCHC	4.4	4	.29	3
Genitor	4.5	5-6	.43	6
I-SGA	6.9	8	.63	8
I-ESGA	4.0	3	.34	5
I-pCHC	3.5	2	.28	2
I-Genitor	3.4	1	.22	1
Cellular	4.5	5-6	.32	4

Algorithm	Avg	rnk	Nrm	rnk
SGA	6.4	7-8	.79	9
ESGA	6.6	9	.63	8
pCHC	4.6	3-4	.41	2
Genitor	4.4	2	.56	6
I-SGA	6.4	7-8	.59	7
I-ESGA	4.8	5-6	.44	3
I-pCHC	4.8	5-6	.46	4
I-Genitor	2.4	1	.26	1
Cellular	4.6	3-4	.47	5

Table 12: Performance on hard problems (left) and long bitstring problems (right)

The non-elitist algorithms (SGA and IslandSGA) clearly perform the worst overall, ranking 8th and 9th in every category. The parallel algorithms seem to perform better on harder functions. Island-Genitor gets the best marks, and the pCHC and cellular genetic algorithms also are consistently good in spite of simplified implementations. It is reasonable to expect a full implementation of CHC to perform better than the pCHC results reported here.

5.4 A Closer Look at Cellular Models

In the previous section, one simple cellular genetic algorithm was tested. It is logical to assume that changes to the topological configuration of a CGA will affect its performance. This section compares the problem-solving performance of several CGA alternatives. The goal is to determine the effectiveness of different CGA topologies, regardless of actual parallel execution.

Three fixed topology CGA's are implemented:

1. *Deme-4.* This algorithm was used in the previous experiments. The best of the four strings above, below, left, and right of an individual is chosen for mating. If either offspring has a higher fitness, the individual is replaced by the most highly fit offspring.

2. *Deme-4+Migration* is identical to Deme-4, but an additional operator, *migration*, is employed which copies one string to a random location at predefined intervals. The use of migration in a CGA was proposed by Gordon *et al.* [GWB92] and Gorges-Schleuter [GrS92], as a way of allowing distant niches to interact.

3. *Deme-12.* Demes consist of the twelve nearest strings. To reduce selective pressure, four strings are chosen randomly from the twelve, and the best of the four is selected for mating. The individual is replaced by the offspring probabilistically according to an adjustable replacement pressure.

Three random walk CGAs are considered:

1. *Walk3.* A deme consists of the strings which reside along a random walk of length 3 starting at the cell in question. Steps along the walk can be up, down, left, or right. The best string along the walk is chosen for mating.

2. *Walk5.* Same as Walk3 (above), but with a walklength of 5.

3. *Walk7.* Same as Walk3 (above), but with a walklength of 7, and probabilistic replacement; the offspring is compared with the individual in question, and the better of the two is replaced according to an adjustable replacement pressure.

In an *Island-CGA*, the grid is divided into smaller subgrids, each of which is a torus that is processed separately using one of the above methods. Migration between subgrids occurs at predefined intervals. In the experiments presented here, each island uses the *Deme-4* algorithm described above.

We run each algorithm on the Rastrigin, Schwefel, and Griewank functions, keeping the following parameters constant: population size = 2500 (50x50), mutation = .005, probability of crossover = 100%, and replacement pressure = 90% (for Deme-12 and Walk7). For the Island-CGA the number of subpopulations is 25, each of which are of size 100 (10x10), and the swap interval is 5 generations. The migration interval for Deme-4+Migration is set to 50 generations. All implementations employ two-point reduced surrogate crossover and next-point mutation. Gray coding is used on the Rastrigin and Griewank functions, but not on the Schwefel function.

The performance of each of the seven CGA's on the the Rastrigin and Schwefel functions is shown in Table 13. For each function, we report the average number of generations it takes for each CGA to find the optimum (averaged over 30 runs), and the standard deviation. Appendix B contains convergence graphs for each algorithm on the three functions.

The performance of each of the seven CGA's on the Griewank function is also shown in Table 13. The algorithms are not always able to solve the problem within 500 CGA generations, so we report the number of runs (out of 30) in which the global optimum is found, along with the average fitness of the best strings found at the end of 500 generations.

Table 14 contains a summary of the ranked performance of the CGA's on each of the functions. Since it is evident that the Schwefel function is the easiest for the CGA's and that the Griewank is the hardest (this concurs with other researchers), we list the functions from left-to-right in increasing order of difficulty.

Average generations to solve (Avg) and standard deviation (Std)				
function	Rastrigin		Schwefel	
algorithm	Avg	Std	Avg	Std
Deme-4	485	95	137	26
Deme-4+mig	485	93	136	26
Deme-12	557	109	126	25
RW-3	606	115	160	31
RW-5	535	102	148	28
RW-7	685	141	158	30
Island-CGA	521	103	152	29

Number solved (ns) and average best (avg).		
	Griewank	
algorithm	ns	avg
Deme-4	20	.009
Deme-4+mig	24	.007
Deme-12	21	.016
RW-3	29	.005
RW-5	25	.006
RW-7	23	.016
Island-CGA	19	.012

Table 13: CGA performance on Rastrigin, Schwefel, and Griewank. Population size = 2500, mutation = .005, crossover probability = 100%, replacement pressure = 90% (where applicable). For Island-CGA, number of subpopulations = 25, subpopulation size = 100 (10x10), migration interval = 5 (generations). For Deme-4+Migration, migration interval = 50. The Griewank function is run for a maximum of 500 generations.

algorithm	Schwefel	Rastrigin	Griewank
Deme-4	3	2	4
Deme-4+mig	2	1	3
Deme-12	1	5	6
RW-3	7	6	1
RW-5	4	4	2
RW-7	6	7	5
Island-CGA	5	3	7

Table 14: Performance of CGAs on Test Suite

CGA's with the most locality (i.e., the smallest fixed demes and the shortest random walks) perform worse on the easiest function (Schwefel) and best on the hardest function (Griewank). The Island-CGA performs rather poorly throughout the suite, which is a surprise considering our earlier results (Section 5.3). Adding a small amount of migration to Deme-4 improved performance on all three problems, but sometimes only slightly. Overall, Deme-4+Migration performed the best of all of the cellular algorithms on this test suite.

6 Conclusions

This paper has introduced a machine-independent way of analyzing parallel genetic algorithms using the Sisal programming language and a dataflow model of computation. Using the dataflow model has made it possible to compare, on equal terms, several parallel genetic algorithms with regards to both problem-solving power and runtime efficiency. The performance of several parallel genetic algorithms has been examined by comparing their ability to solve hard function optimization problems, and by using the dataflow simulator to generate their parallelism profiles.

Examination of the Sisal code alongside the parallelism profiles has helped reveal new areas where parallelism could be exploited (especially in SGA and Genitor). Parallelism exists to varying degrees among all of the genetic algorithms. At the lowest level, parallelism is sometimes exploitable by vectorization. String-level parallelism is equivalent in all of the algorithms except Genitor, which has the least. However, the parallelism that Genitor does contain is fully exploitable, whereas this is not the case, say, for SGA, due to overhead.

Using the dataflow model made it possible to identify several potentially removable bottlenecks in the algorithms. The worst bit-level bottlenecks found were in the two-point reduced surrogate operator, and the DeGray operator. DeGray is probably not a serious bottleneck in real applications, but two-point crossover is more of a problem. At courser levels of granularity, algorithmic bottlenecks and/or communication overhead necessitate the use of cellular or island models, and measurements confirm that bottlenecks in these models are fully removable on a parallel architecture.

The relationship between parallelism and problem-solving power was examined by comparing several parallel genetic algorithms across a wide range of optimization functions. The results show that speedup due to parallelism is not offset by declines in problem-solving capabilities. In fact, parallel genetic algorithm using some form of restricted selection and mating based on locality that are executed serially yield better performance than single population implementations with global "panmictic" mating. Alternative population structures such as cellular, steady-state (i.e., Genitor), and CHC approaches are at least as effective as elitist (and non-elitist) versions of the standard "Holland-style" genetic algorithm. However, combining island and cellular approaches seems to have a negative effect. The study also confirms that elitist strategies perform better than non-elitist ones, which has been reported previously by other researchers. The results do not address Holland's original contention that SGA displays superior performance with respect to the average fitness of all strings throughout a run (a measure which he referred to as *online* performance).

Locality also affects problem-solving power. Cellular algorithms with high locality (small fixed demes or short walks) perform better than those with low locality on our hardest test

function. Adding migration to a cellular model also results in slightly improved performance on each of the test functions.

Finally, the machine-independent nature of this study has led to the development of some new genetic algorithms: pCHC, CGA+Migration, Island-pCHC, Island Cellular, and PreGenitor. Of these, Island Cellular is the least important, but the others represent improvements over particular previously existing algorithms, either in terms of parallelism or problem-solving power.

Two areas of future work would strengthen our research. First, we need a more rigorous measure of locality. Periodically we have had to use fairly subjective means to express the effects of communication overhead on the performance of a parallel implementation. The development of a locality metric would allow actual speedup and/or overhead estimates to be quantified for various scenarios. Some simple metrics have been proposed (see [Gor94a]), but they are still preliminary in nature. Second, it would naturally be satisfying to realize more of the new algorithms on real hardware.

The approach introduced in this paper opens the door to further empirical studies of parallel genetic algorithm implementations via common measures. The use of Sisal and the dataflow model offers flexibility without sacrificing the capacity for high performance. Perhaps by continuing to study genetic algorithms from a machine-independent *algorithms* perspective, researchers will discover even better ways of improving their performance.

References

[Böh90] A. Böhm. Instrumenting Dataflow Systems. *Instrumentation of Parallel Computing Systems 2*, 1990

[CJ91] R. Collins and D. Jefferson. Selection in Massively Parallel Genetic Algorithms. *Proceedings of the 4th International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1991

[Dav91] Y. Davidor. A Naturally Occurring Niche & Species Phenomenon: The Model and First Results. *Proceedings of the 4th International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1991

[DeJ75] K. DeJong. An Analysis of the Behavior of a Class of Genetic Adaptive Systems. PhD thesis, University of Michigan, 1975.

[Esh90] L. Eshelman. The CHC Adaptive Search Algorithm. *Foundations of Genetic Algorithms and Classifier Systems*, Morgan-Kaufmann, 1991

[Fis58] R. Fisher. *The Genetical Theory of Natural Selection.* Dover, New York, 1958.

[Gol89a] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning* Addison-Wesley, ©1989

[GKD89] D. Goldberg, B. Korb, and K. Deb. Messy Genetic Algorithms: Motivation, Analysis, and First Results. *Complex Systems 3*, 1989.

[Gol89b] D. Goldberg. Genetic Algorithms and Walsh Functions: Part II, Deception and its Analysis. University of Alabama, *TCGA No. 88005*, 1989.

[Gol90] D. Goldberg. A Note on Boltzmann Tourn. Selection for Genetic Algorithms and Population-Oriented Simulated Annealing. University of Alabama, *TCGA No. 90003*, 1990.

[GD91] D. Goldberg and K. Deb. A Comparative Analysis of Selection Schemes used in Genetic Algorithms. *Foundations of Genetic Algorithms and Classifier Systems*, Morgan-Kaufmann, 1991

[GWB92] V. Gordon, D. Whitley, and A. Böhm. Dataflow Parallelism in Genetic Algorithms. *Parallel Problem Solving from Nature 2*, North Holland, 1992

[GW93a] V. Gordon and D. Whitley. Serial and Parallel Genetic Algorithms as Function Optimizers. *Proceedings of the 5th International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1993.

[GBW94] V. Gordon, A. Böhm and D. Whitley. A Note on the Performance of Genetic Algorithms on Zero-One Knapsack Problems. *Symposium on Applied Computing (SAC'94), Genetic Algorithms and Combinatorial Optimization*, 1994

[Gor94a] V. Gordon. Locality in Genetic Algorithms. *The First IEEE Conference on Evolutionary Computation (WCCI)*, IEEE, 1994.

[Gor94b] V. Gordon. Asynchronous Parallelism in Steady-State Genetic Algorithms. *Neural and Stochastic Methods in Image and Signal Processing III (SPIE)*, 1994.

[GSc89] M. Gorges-Schleuter. ASPARAGOS – An Asynchronous Parallel Genetic Optimization Strategy. *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1989

[GrS91] M. Gorges-Schleuter. Explicit Parallelism of Genetic Algorithms through Population Structures. *Parallel Problem Solving from Nature*, Springer Verlag, 1991.

[GrS92] M. Gorges-Schleuter. Comparison of Local Mating Strategies in Massively Parallel Genetic Algorithms. *Parallel Problem Solving from Nature 2*, North Holland, 1992.

[GKB87] J. Gurd, C. Kirkham, and W. Böhm. The Manchester Dataflow Computing System. in J. Dongarra, *Experimental Parallel Computing Architectures*, Special Topics in Supercomputing 1, North Holland, 1987.

[Hol75] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, ©1975.

[MW94] K. Mathias and D. Whitley. Transforming the Search Space with Gray Coding. *Proceedings of the First IEEE Conference on Evolutionary Computation*, IEEE, 1994.

[McG85] J. McGraw *et al.* SISAL - Streams and Iteration in a Single-Assignment Language. *Lawrence Livermore National Laboratory M-146*, 1985.

[MSB91] H. Mühlenbein, M. Schomisch, and J. Born. The Parallel Genetic Algorithm as Function Optimizer. *Proceedings of the 4th International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1991.

[Müh91a] H. Mühlenbein. Evolution in Time and Space – The Parallel Genetic Algorithm. *Foundations of Genetic Algorithms and Classifier Systems*, Morgan-Kaufmann, 1991

[Müh91b] H. Mühlenbein. Asynchronous Parallel Search by the Parallel Genetic Algorithm. *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, IEEE, 1991.

[Müh92] H. Mühlenbein. How Genetic Algorithms Really Work I: Mutation and Hillclimbing. *Parallel Problem Solving from Nature 2*, North Holland, 1992

[MS93] H. Mühlenbein and D. Schlierkamp-Voosen. Predictive Models for the Breeder Genetic Algorithm: Continuous Parameter Optimization. *Evolutionary Computation*, vol. 1, 1993.

[SWM91] T. Starkweather, D. Whitley, and K. Mathias. Optimization Using Distributed Genetic Algorithms. *Parallel Problem Solving from Nature*, Springer Verlag, 1991.

[Tan89] R. Tanese. Distributed Genetic Algorithms. *Proceedings of the 3rd International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1989

[WK88] D. Whitley and J. Kauth. GENITOR: a Different Genetic Algorithm. *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, Denver, 1988

[WS90] D. Whitley and T. Starkweather. Genitor II: a Distributed Genetic Algorithm. *Journal of Experimental and Theoretical Artificial Intelligence*, 1990.

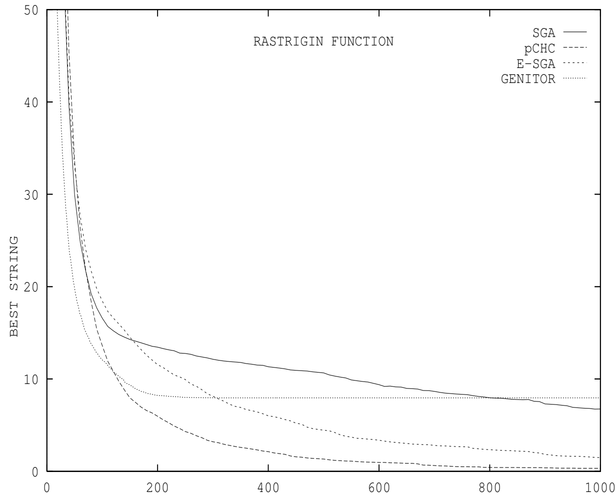
[Whi91] D. Whitley. Fundamental Principles of Deception in Genetic Search. *Foundations of Genetic Algorithms and Classifier Systems*, Morgan-Kaufmann, 1991

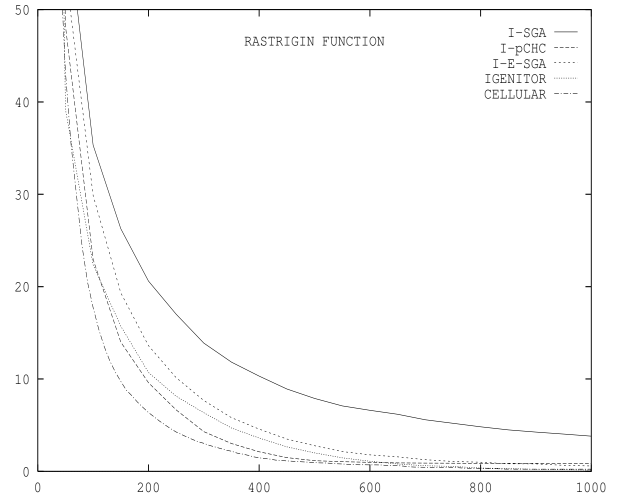
[WDC92] D. Whitley, R. Das, and C. Crabb. Tracking Primary Hyperplane Competitors During Genetic Search. *Annals of Mathematics and Artificial Intelligence*, 1992.

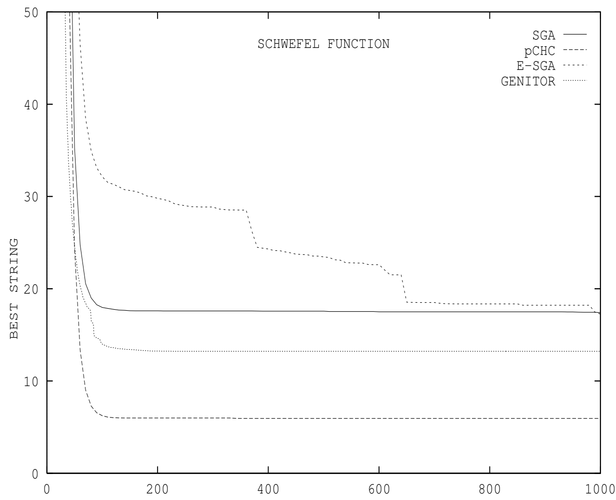
[Whi93] D. Whitley. Cellular Genetic Algorithms. *Proceedings of the 5th International Conference on Genetic Algorithms*, Morgan-Kaufmann, 1993.

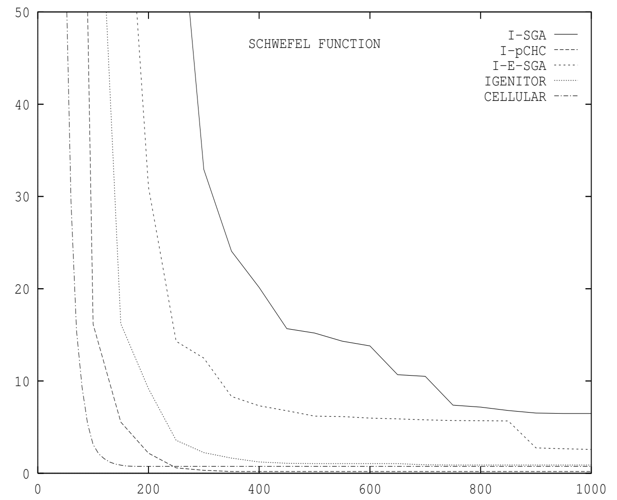
[Wri32] S. Wright. The Roles of Mutation, Inbreeding, Crossbreeding, and Selection in Evolution. *Proceedings of the 6th International Congress on Genetics*, 1932

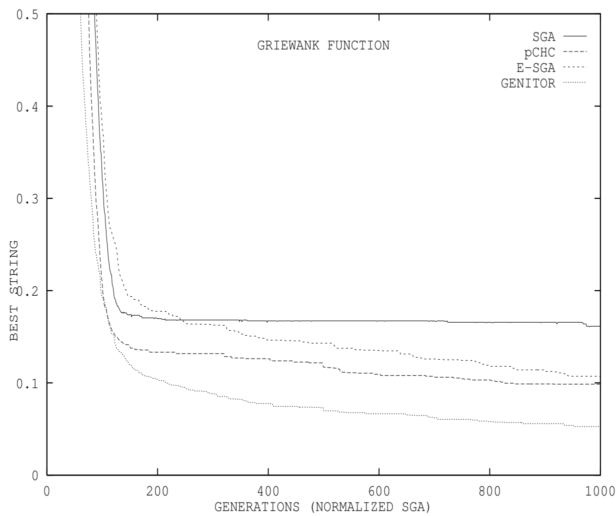
Appendix A – Convergence Graphs

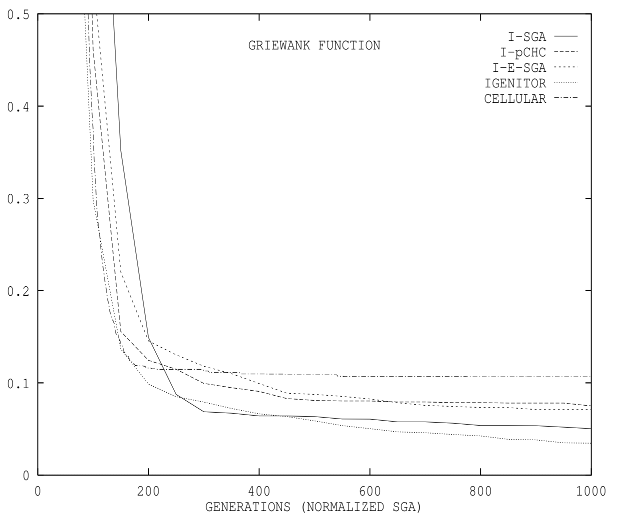












Appendix B – Cellular Models