Dataflow parallelism in genetic algorithms

V. Scott Gordon, Darrell Whitley, and A. P. Wim Böhm
Computer Science Department, Colorado State University, Fort Collins, CO 80523 USA
gordons/whitley/bohm@cs.colostate.edu

Abstract

Parallel genetic algorithms are examined in a machine-independent and application-independent fashion. There are three major potential benefits of this approach. First, it encourages algorithm development independent of machine architecture. Thus, we reexamine *what kind of* parallel genetic algorithms are possible. Second, we examine the potential parallelism in various implementations of parallel genetic algorithms. Implicit parallel programming languages such as Sisal allow us to generate parallelism profiles and critical path information by executing in a dataflow environment. This allows a comparative evaluation of different genetic algorithms using a single parallel model of computation. Third, algorithms implemented in Sisal are more portable and more scalable than algorithms explicitly mapped onto parallel computer architectures.

1 INTRODUCTION

Previous research in parallel genetic algorithms has been driven by available hardware or other *ad hoc* considerations. Typically, the programmer must direct the distribution of tasks and data in an explicit fashion over a particular parallel architecture or architectural model. This approach is necessarily machine dependent and engenders the disadvantages of other *ad hoc* approaches: lack of portability, lack of scalability, and lack of generality. An alternative way of studying potential parallelism is by using implicit parallel programming languages such as Sisal [13]. These languages allow one to analyze the potential parallelism in a program by executing it in a dataflow simulator. They also allow programs to be ported to different parallel machines without modification. Sisal has been implemented on a variety of parallel machines and displays single-processor efficiency competitive with C or Fortran [2].

We start by coding a "standard" serial genetic algorithm [9] in Sisal and looking at its execution characteristics on a simulator of the Manchester Dataflow Machine [8]. The simulator generates parallelism profiles and various measures such as critical path length (total time steps required) and average parallelism. We then do the same for other parallel genetic algorithm models which have been proposed by various researchers. This approach 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.

Eventually, this research will be coupled with analysis of convergence properties of the algorithms in order to offer meaningful conclusions regarding the speedup which can be achieved by parallelizing genetic algorithms. However, at this point in time, formal convergence properties for different forms of the genetic algorithm are not well understood. Most of the current work on convergence of genetic algorithms is done empirically, and results are often application-dependent. Thus, in this paper we do not try to determine what types of parallel genetic algorithms display the best problem solving capabilities. Rather, our work focuses on understanding the potential parallelism of genetic algorithms independent of any specific machine or application.

2 EXISTING RESEARCH IN PARALLEL GENETIC ALGORITHMS

One choice that must be made when implementing parallel genetic algorithms is whether a single population will be used with global random mating and replacement, or whether some form of locality will be imposed on mating and replacement strategies.

The same issue is at the heart of the debate between Wright and Fisher concerning the role of spatial structure in natural genetic populations. Wright claimed that modeling biological evolution requires considering local interaction and spatial isolation [21]. These views were challenged by Fisher, who believed that biological evolution could be

modeled as having unlimited interaction between individuals [6]. These processes are called *local* and *panmictic* interaction, respectively. Each has been used as biological justification for various parallel genetic algorithms [3, 15].

Current literature on parallel genetic algorithms can be separated into three categories: *global population models*, *island models*, and *massively parallel* genetic algorithms. We examine each in turn.

Global Population Models. Serial genetic algorithms such as SGA [10], Genitor [20] and CHC [5] typically employ a single population without locality considerations. Some work has been done in parallelizing these algorithms directly. For example, Genitor can be parallelized asynchronously on a shared-memory machine by allowing multiple processes to simultaneously access a single shared population. Fogarty has described results with similar methods [7]. Parallelism of SGA has been discussed by Goldberg [10]. Ironically, the parallel implementation of global population models has not received as much attention as other approaches. This is in part due to the fact that machine dependent considerations lead implementors toward other approaches.

Island Models. This model was developed for use on machines with a relatively small number of processors (e.g. less than 1,000). 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 [18, 19, 20]. Many researchers have reported excellent results with this approach. For example, Whitley has shown improved empirical results on several problems over single population genetic algorithms even when the total number of strings is the same [18, 20]. Tanese has proposed that the island model is further enhanced when different subpopulations utilize different parameter settings [19]. Mühlenbein characterizes the island model as being in agreement with Wright's model [15].

Massively Parallel Models. Massively parallel models are designed to take advantage of machines with a large number of processors (1000 or more). In the simplest case one can use a single large population with one string per processor. In order to avoid high communication overhead, these algorithms generally impose some limitation on mating based on distance. For example, strings may be arranged conceptually on a grid, and they may only be allowed to perform crossover with adjacent strings on the grid. This leads to a form of locality known as "isolation-by-distance." It has been shown that neighborhoods of similar individuals (also called *demes*) typically form, and that these neighborhoods function as "niches" in ways that may be similar to discrete subpopulations. However, there are no actual boundaries between neighborhoods and demes can more easily be overtaken and exterminated by competing demes than in the island model. Nevertheless, excellent results have been reported by various researchers [4, 14, 15].

3 THE DATAFLOW APPROACH AND SISAL

As we have seen, previous research in parallel genetic algorithms has been driven by available environments or other *ad hoc* considerations. The resulting lack of portability makes it difficult to analyze and compare independent results *empirically*. We propose instead to analyze parallel genetic algorithms independent of environmental considerations.

The attractiveness of the dataflow model of computation is that all forms of parallelism can be expressed in it. This makes dataflow an ideal environment for the analysis of parallel algorithms which leads to a better understanding of an algorithm's inherent parallelism, its sequential threads, and its resource requirements [1].

Sisal is a functional side-effect-free language [13]. There are a number of advantages to using Sisal to write parallel programs. Sisal exposes parallelism implicitly because the lack of side-effects allows for highly optimizable compilation into parallel code. Sisal is portable across a wide 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.

3.1 Parallel SGA

Goldberg's Simple Genetic Algorithm (SGA) [9] is a well known variation on Holland's original model [11]. Rather than describe SGA here, we describe key aspects of our Sisal implementation. When implementing traditional simple genetic algorithms, global information is often required, such as the *average fitness* of the strings in the population. An alternate way of allocating reproductive opportunities is by using a *tournament* [10]. For example, in one implementation of tournament selection, two strings are randomly sampled. The string with the higher fitness is then passed into the intermediate generation. This process continues until the intermediate generation is filled. We can then expect the best string to have two copies in the intermediate generation, the average string to have one copy, and the worst string to have no copies, thus creating a linear ranking.

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 we rebuild the whole population, and 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 we utilize $n/2$ processes (where n is the population size), each of which generates two members of the next generation. Each process can also perform the selection, crossover (if necessary), and mutation.

We divide the population into two interleaved halves, denoted *even* and *odd*. For each pair of slots in the population we select four individuals from the previous generation. Each slot conducts a tournament, keeping the best of two strings. Each process thus generates two elements, and these pairs are collected to form a single new population.

Probability of crossover is simulated by not performing crossover on the intermediate generation beyond a fixed point. 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. Our implementation of SGA is shown in Figure 1. Note that crossover does not occur below the heavy line.

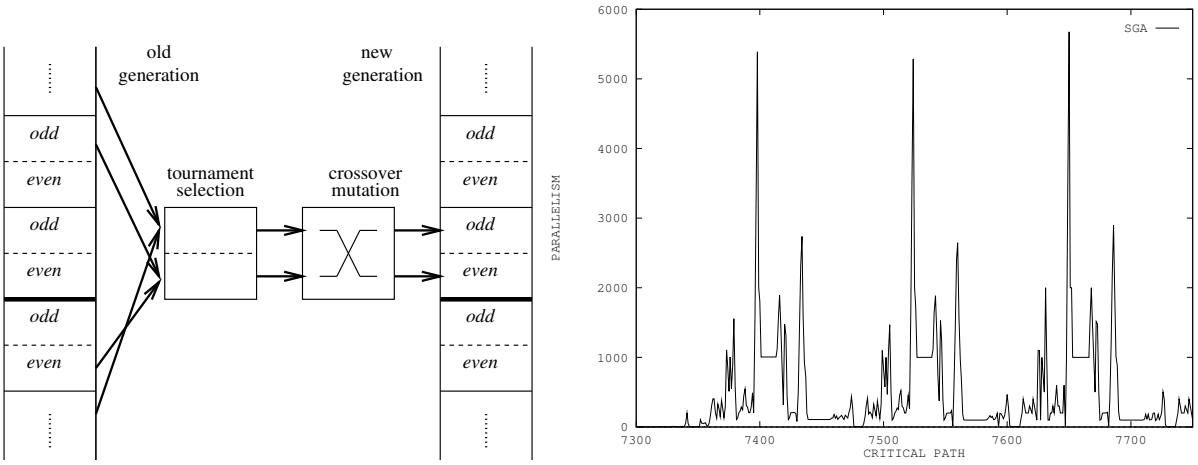

Figure 1: Parallel SGA: algorithm and parallel profile.

In Figure 1, we see a parallelism profile of three generations of SGA with a population size of 100. 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. For example, since mutation is a highly parallel activity (theoretically every bit can be mutated in parallel), mutation is seen as a tall "spike" in the graph.

3.2 Island model

The island model involves running several single population genetic algorithms in parallel. In the implementation used here, each "island" is an SGA with a subpopulation. Generation-to-generation processing is done serially. Within a generation, each subpopulation is rebuilt using SGA as described earlier. Information is shared between islands by allowing strings to migrate from one island to another. Migration is characterized by *migration topology*, *migration rate*, and *number of strings* involved in the migration.

In our implementation, the migration topology is a *ring*; strings are only allowed to migrate to the next subpopulation in the ring. We have chosen a simple migration routine that works as follows. First, a tournament is held for each subpopulation. The losing individual is replaced by the winning individual from the adjacent subpopulation. The use of tournament methods here means that we are not necessarily migrating the very best individuals from each subpopulation, but the strings tend to be above average. Larger tournaments increase the expected fitness of the strings that migrate and decrease the expected fitness of the strings that are replaced.

Note that this implementation of the island model is somewhat non-standard, because we have employed tournament selection to determine which individuals will migrate. Also note that Sisal will exploit whatever parallelism is available, both within and between islands. In Figure 2, we see a parallelism profile of the Island algorithm on four subpopulations each of size 4. Migration occurs between the second and third generations.

3.3 Massively parallel model

As described earlier, massively parallel genetic algorithms typically involve one individual per node (processor), where mating is limited to a deme (neighborhood) near the node. Generation-to-generation processing is done serially. The processing of a single generation, however, is again highly parallelizable. In this case each string in the new population can be generated in parallel. Each process gathers neighboring individuals and performs the standard operations.

For our first implementation, we consider only the four individuals directly above, below, left, and right of the node. We select the best of these four, and perform crossover with the individual at the node in question. If the resultant offspring is better than the original, the offspring is kept. Otherwise the original individual is kept. Edge elements wrap around, forming a torus. Population sizes are a perfect square.

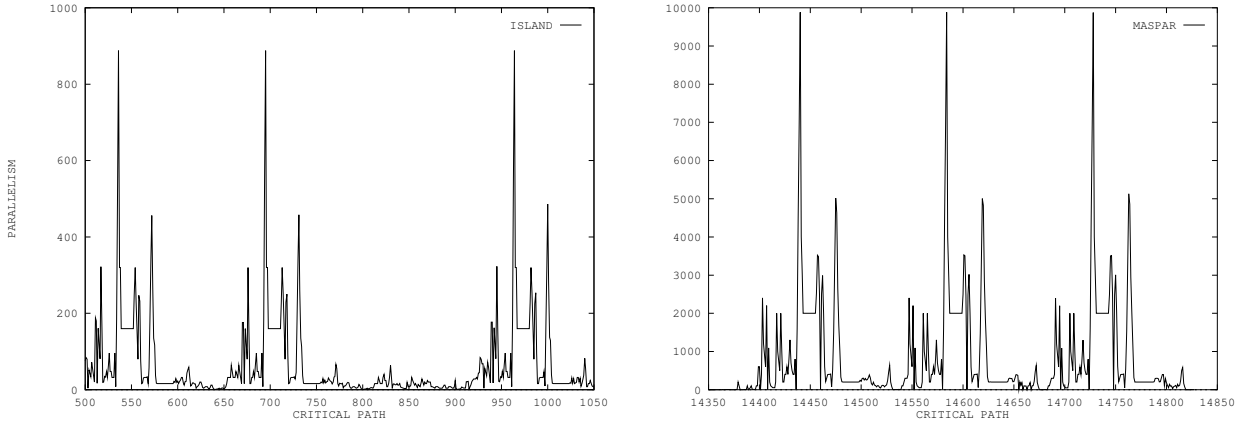

Figure 2: Profiles for Island (left) and Massively parallel (right) models.

In Figure 2, we see a parallelism profile of three generations of the massively parallel algorithm on population size of 100.

3.4 Average parallelism and critical path

Sisal programs can be evaluated using a dataflow simulator which collects statistics while it executes the code. Simulation proceeds in discrete *time steps* during which all enabled instructions (instructions for which all 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. 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*). The parallelism of the program is graphed in a parallelism profile by plotting the number of executed instructions at each time step.

Average parallelism is defined as S_1/S_∞. The dataflow simulator reports values for average parallelism, S_1, and S_∞ at the conclusion of program execution. However, these figures are for runs which include initialization. A more interesting measure is the average parallelism for a single generation. We obtain this figure by running each algorithm for 4 generations and then for 3 generations, and calculating the ratio of the differences:

$$S_1/S_\infty = (S_1^4 - S_1^3)/(S_\infty^4 - S_\infty^3) \tag{1}$$

We can examine the parallelism profiles for various parameter settings to determine the sensitivity of the algorithm to a particular parameter.

Table 1 presents average parallelism and critical path information for *one generation* for SGA, the Island model and massively parallel model. In the case of the Island model, these numbers are for the combination of one generation and one migration.

We can clearly see that 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 larger population sizes, this is an encouraging result.

Table 1: Average parallelism metrics and critical path length

	PopSize	Avg parallelism	Critical path	Instr executed
SGA	100	658	137	81227
SGA	16	109	137	14996
Island	(20x5) 100	351	275	96731
Island	(4x4) 16	66	275	18326
Mass.Par.	100	1061	155	164510
Mass.Par.	16	171	155	26505

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

Since we are using a dataflow simulator to generate these statistics, there are implicit assumptions regarding machine characteristics. In particular, we assume that we can utilize parallelism at an arbitrarily fine grained level. However, we cannot expect to have such an ideal environment. On a real machine, some of our genetic operators would be implemented at a courser grain. For example, we use a mutation operator that mutates every bit 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, we would apply a different mutation operator that might be sensitive to the population size or string length.

Parallelism profiles allow us to isolate key operators in terms of their impact on parallelism. For example, in Figure 3 we see a single generation of SGA in closer detail, both with mutation turned on and mutation turned off. Since mutation is theoretically a highly parallel activity, we note the loss of the very tall "spike" in the graph.

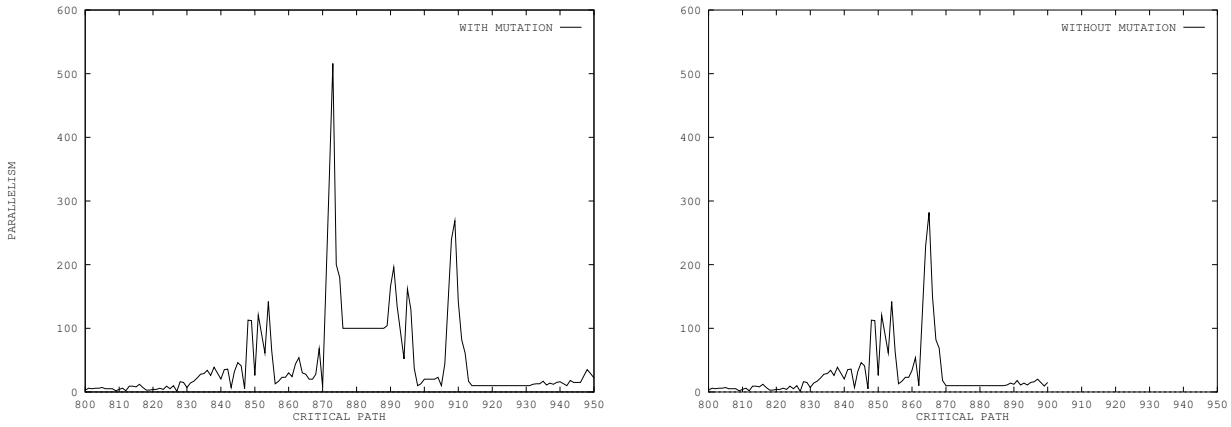

Figure 3: SGA: one generation with and without mutation.

4 EXISTING METHODS – A MACHINE INDEPENDENT LOOK

Our work with the dataflow computing model suggests that it may be more useful to categorize existing work on parallel genetic algorithms by algorithmic properties rather than hardware constraints. Categorizing parallel genetic algorithms by algorithmic properties reveals a number of implementations that have not yet been attempted.

Three central aspects characterizing genetic algorithms are *string distribution* (single population, subpopulations, or one string per processor), *migration topology* (either local or panmictic), and *mating topology* (local or panmictic). For an implementation of these algorithms to be effective the mapping from the algorithmic characteristics onto the *machine topology* (such as a ring or grid) has to fit naturally.

Some categories that have already been explored are:

- One string per processor with local mating and migration, and either a ring or grid topology. This describes the existing massively parallel model [3, 4, 14].
- Panmictic mating within subpopulations and a grid or cube migration topology. This describes the existing "stepping-stone" model [14, 16, 17, 19].
- Panmictic mating within subpopulations and a fully-connected migration topology. This describes the existing island models [12, 20]. As the algorithm assumes a fully-connected migration topology, a successful mapping is only possible on an architecture with a small number of processors.
- Single population (and therefore no migration) with panmictic mating. This describes parallelization of standard genetic algorithms [7, 10]. As the algorithm assumes fully-connected algorithmic topology without subpopulations, a successful mapping is only possible on a shared memory machine.

Some categories that have not yet been explored are:

- One string per processor with local mating and panmictic migration, using a machine topology such as a ring or grid. This describes a massively parallel genetic algorithm with periodic random migration. The use of panmictic migration is interesting because it models the notion of periodic interaction between distant demes. The use of panmictic migration over a fixed machine topology could be feasible if migration does not happen very often.
- Subpopulations with local mating within each subpopulation and a fully-connected panmictic migration topology. Each subpopulation is implemented as a massively parallel model, which fits naturally on a fully-connected small set of SIMD machines.

Here we see two quite reasonable algorithms for which results have not yet been published. We see that viewing existing parallel genetic algorithm research in this way reveals the hardware-architecture driven nature of current research. Our implicit parallel method gives us a way of quickly developing and evaluating new models.

4.1 An example of a new parallel model: parallel CHC

In its original form, CHC [5] is similar to SGA except that the best n strings from both the original generation and next generation are extracted to form the new generation in CHC. There are additional changes such as restarts and incest prevention, but in our version we concentrate only on preserving the elitism that CHC introduces into SGA.

Selecting the best n from $2n$ strings involves a global operation equivalent to sorting. Here again it is more efficient to employ tournament selection. We approximate the effect of CHC by having each iteration compare its two newly-produced children with two predetermined previous elements (based on iteration number), and output the best two of the four. An illustration of this process is given in Figure 4. Note that this algorithm does not guarantee the best n out of the $2n$ individuals, but it does guarantee that at least the best *two* individuals will survive.

Figure 4 shows a parallelism profile for CHC on population size of 100. Table 2 shows average parallelism and critical path information for population sizes of 16 and 100.

Table 2: Parallelism metrics for the CHC algorithm

	PopSize	Avg parallelism	Critical path	Instr executed
CHC	100	520	185	96353
CHC	16	86	184	15754

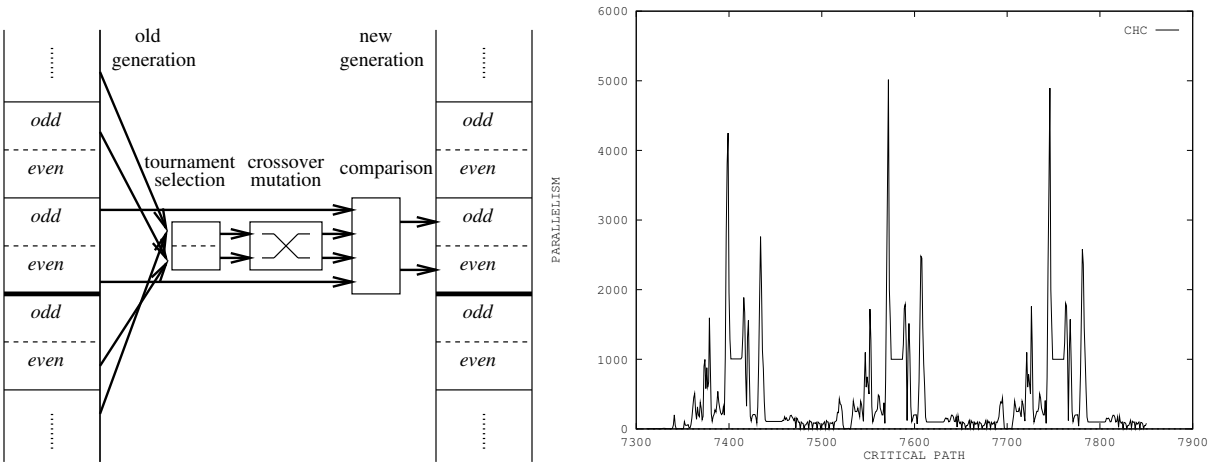

Figure 4: Parallel CHC: algorithm and parallel profile.

5 DISCUSSION AND CONCLUSIONS

One advantage of using Sisal is that we can port our code to several architectures without modifying it. Sisal runs on single-processor workstations and on several parallel architectures. Thus, we can exploit parallelism without worrying about the underlying architecture. This means that implementors can pick algorithms based on effectiveness instead of worrying about how well they map to a particular architecture. This makes the resulting implementations more portable, but it also means they are more scalable.

Dataflow simulation makes it clear that there are varying degrees of parallelism in genetic algorithms at various points in the computation. Such information is very useful from an algorithm development perspective. Examination of the Sisal code alongside the parallelism profile may help reveal where parallelism is not being exploited.

It is remarkable that the parallelism profiles look extremely similar, regardless of which genetic algorithm is being modeled. This suggests that while implementors may be doing a good job of mapping the algorithms to architectures, *they have actually done very little to change the nature of the underlying parallelism which is inherent in the genetic algorithm.* The algorithms differ in the way their topology expresses locality and exploits machine architecture. A shortcoming of our simulations is that we have assumed a uniform memory access time. In reality, for MIMD machines the movement of data involves overhead inversely proportional to the degree of locality in the algorithm. While SGA appears here to be more efficient in terms of total instructions executed, the massively parallel algorithm is a more efficient algorithm in terms of locality. Therefore, the massively parallel model would be more appropriate for a SIMD environment than SGA. In future work we will enhance the simulations to take locality issues into account.

Further work is also needed in terms of testing new and existing algorithms, both in terms of effectiveness and in terms of evaluating their potential parallelism. A very fast, highly parallelizable genetic algorithm is of little use if it fails to produce acceptable solutions. We particularly need to look at how the changes we have made to these algorithms impacts their effectiveness.

We have introduced a machine-independent way of analyzing parallel genetic algorithms using the Sisal programming language and a dataflow model of computation. We have presented specific Sisal implementations of some parallel genetic algorithms and examined their performance by using the dataflow simulator to generate parallelism profiles and measures for average parallelism and critical path length. This approach opens the door to empirical study of different parallel genetic algorithm implementations via common measures, while retaining the capacity for high performance.

References

[1] A. Böhm. Instrumenting Dataflow Systems. *Instrumentation of Parallel Comp Sys 2*, 1990

[2] D. Cann. Retire Fortran? A Debate Rekindled. *SuperComputing 91*, IEEE Comp Soc Press

[3] R. Collins and D. Jefferson. Selection in Massively Parallel Genetic Algorithms. *Proc. 4th International Conf. on Genetic Algorithms*, Morgan-Kaufmann, 1991

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

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

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

[7] T. Fogarty. Implementing the Genetic Algorithm on Transputer Based Parallel Processing Systems. *Parallel Problem Solving from Nature*, Springer Verlag, 1991

[8] J. Gurd, C. Kirkham, and W. Böhm. The Manchester Dataflow Computing System. in *Experimental Parallel Comp. Arch.*, Topics in Supercomputing 1, North Holland, 1987

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

[10] D. Goldberg. A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing. Univ of Alabama, TCGA No. 90003 May 1990

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

[12] G. Liepens and S. Baluja. apGA: An Adaptive Parallel Genetic Algorithm *unpublished*, 1991

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

[14] H. Mühlenbein, M Gorges-Schleuter, and O. Kramer. Evolution Algorithms in Combinatorial Optimization. *Parallel Computing 7*, North Holland, 1988. pp 65-85

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

[16] C. Pettey, M. Leuze, and J. Grefenstette. A Parallel Genetic Algorithm. *Genetic Algorithms and their Applications*, Morgan-Kaufmann, 1987

[17] P. Spiessens and B. Manderick. A Massively Parallel Genetic Algorithm – Implementation and First Analysis. *Proc. 4th Int Conf on Genetic Algorithms*, Morgan-Kaufmann, 1991

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

[19] R. Tanese. Distributed Genetic Algorithms. *Proc. 3rd International Conf. on Genetic Algorithms*, Morgan-Kaufmann, 1989

[20] D. Whitley and T. Starkweather. Genitor II: a Distributed Genetic Algorithm. *Journal Expt. Theor. Artif. Intell 2*, 1990. pp 189-214

[21] S. Wright. The Roles of Mutation, Inbreeding, Crossbreeding, and Selection in Evolution. *Proc. 6th Int. Congr. on Genetics*, 1932