A DISTRIBUTED GENETIC ALGORITHM WITH MIGRATION FOR THE DESIGN OF COMPOSITE LAMINATE STRUCTURES

MATTHEW T. MCMAHON and LAYNE T. *WATSON

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061-0106, USA

This paper describes the development of a general Fortran 90 framework for the solution of composite laminate design problems using a genetic algorithm (GA). The initial Fortran 90 module and package of operators result in a standard genetic algorithm (sGA). The sGA is extended to operate on a parallel processor, and a migration algorithm is introduced. These extensions result in the distributed genetic algorithm with migration (dGA). The performance of the dGA in terms of cost and reliability is studied and compared to a sGA baseline, using two types of composite laminate design problems. The nondeterminism of GAs and the migration and dynamic load balancing algorithm used in this work result in a changed (diminished) workload, so conventional measures of parallelizability are not meaningful. Thus, a set of experiments is devised to characterize the run time performance of the dGA.

KEY WORDS: Genetic Algorithms, Composite laminate design

C.R. CATEGORIES: C.1.2, D.1.3, G.1.6, J.2

NOMENCLATURE

A	average number of optima found per run for a set of GA runs
C_n	normalized cost for a set of GA runs
dGA	distributed GA
EL	elitist selection
ME1	multiple elitist 1 selection
ME2	multiple elitist 2 selection
N_g	average number of generations to convergence for a set of GA runs
N_x	strain in the x direction
N_y	strain in the y direction
N_{xy}	strain in the xy plane
n_k	the number of individuals retained by ME selection strategies
n_m	maximum number of migrants in a single mutation
n_s	actual number of migrants in a single migration
P	subpopulation size
p	the number of processors used in a distributed computation
p_a	probability of gene addition
p_c	probability of crossover
p_d	probability of gene deletion
p_i	the ith processor in a distributed computation
p_m	probability of migration
p_{mm}	probability of material gene mutation
p_{mo}	probability of orientation gene mutation
p_p	probability of gene permutation

* Author to whom correspondence should be addressed

p_s	probability of gene swap
R	apparent reliability of a set of GA runs
S	the number of subpopulations in a set of GA runs
sGA	standard GA
t_d	execution time using the dynamic load balancing algorithm
t_s	execution time using the static load balancing algorithm
V	the genetic alphabet for a particular problem
V_m	the genetic alphabet for material genes
V_o	the genetic alphabet for orientation genes
y	a real-valued random number

1. INTRODUCTION

Numeric optimization problems are often solved using continuous techniques. The problem of composite laminate stacking sequence optimization has been formulated as a continuous design problem and solved using gradient based techniques [12], but these methods of solution are not always successful, for two reasons. First, stacking sequence design often involves discrete design variables, such as ply thickness and orientation, which must be converted to continuous variables for solution. Converting continuous solutions back to discrete allowable values often results in infeasible or sub-optimal designs. Second, composite laminate design problems often have discontinuous or noisy objective functions, or more than one global solution. The genetic algorithm (GA) is an alternative optimization method which can search through design spaces and deal with noisy and discontinuous objective functions, using discrete encoding of the parameters of the problem being optimized. Research has shown GAs to be amenable to the solution of composite laminate design problems (e.g., [12], [19].

Genetic algorithms employ Darwin's concept of natural selection by creating a population of candidate designs and applying probabilistic rules to simulate the evolution of the population [6]. Individuals in the population are discrete encodings of candidate solutions to the problem being solved, and the evolutionary process searches for optimal designs using payoff (objective function) information only. GAs are less likely than conventional optimization techniques to get trapped in locally optimal areas of the search space, and the GA's population structure makes it useful in exploring many candidate designs in parallel.

As powerful parallel computers become more accessible to researchers, it becomes more feasible to harness their power for use with GAs. The GA is inherently parallel, through its population structure, and different parallel algorithms have been introduced to take advantage of this aspect (see, for example, [28], [29]. One benefit of a parallel GA implementation is that further exploitation of genetic information is made possible, through migration. Migration is an extension to the standard genetic algorithm in which otherwise separately evolving subpopulations occasionally identify and exchange genetic information. This paper explores the effect of migration on the GA's performance, within the domain of composite laminate structure design.

A Fortran 90 GA framework is developed for use with composite laminate design optimization. This framework includes a GA module, encapsulating GA data structures and basic operations, and a package of GA operators, which work with the module. A standard GA (sGA) is designed using the Fortran 90 GA framework, and two composite laminate design problems are tested. The first test problem has one global solution amongst many near-optimal solutions, while the second test problem contains several globally optimal solutions. Performance measures—namely normalized cost C_n and reliability R—for the sGA are established and reported for these two test problems. These performance measures serve as a baseline for the performance of the GA.

An extension to the Fortran 90 GA framework is introduced. The distributed GA (dGA) extension implements a migration algorithm to work with a group of subpopulations evolving in parallel. The goal of the migration algorithm is to improve the normalized cost and reliability of a set of GA optimization runs without significantly impacting the time it takes to make those runs. The parallel dGA is implemented as a fully distributed algorithm with dynamic load balancing and termination detection. Sophisticated distributed control techniques are especially effective for nondeterministic algorithms with highly variable workloads, such as the dGA.

Two sets of experiments are conducted using the dGA. In the first set of experiments, the effect of the dGA on normalized cost and reliability is established, and the performance of the dGA is compared to that of a single population of varying size.

Nondeterministic algorithms coupled with dynamic load balancing and distributed control, as found in the dGA, result in variable parallel workloads, making parallel performance evaluation complicated. The second set of experiments examines the performance of the distributed algorithm itself. First, the effective parallelizability of the dGA is examined. Next, the effect on run time of varying the migration rate is observed. Finally, the performance of the dynamic load balancing algorithm is compared to that of a static load balancing algorithm.

The dGA is found to improve both the reliability and the normalized cost for the test problems explored here. In addition, the migration algorithm's effect on execution time is found to be small relative to the improvement in normalized cost. Finally, the experiments designed to explore the parallel performance of the dGA demonstrate linear effective parallelizability for constant work.

2. GENETIC ALGORITHMS

As early as the 1950's, the biological metaphor of evolution was being applied to computation (e.g., [1]-[3]). As computational power has increased and the foundations of these evolutionary algorithms have been formalized and improved, they have increasingly been used in the solution of optimization problems. Currently, Bäck [4] identifies three strongly related but independently developed approaches to evolutionary computation: genetic algorithms, evolutionary programming, and evolution strategies. Although identified as distinct aspects of evolutionary programming, the three evolutionary strategies are closely related by their mimicry of natural evolutionary processes. The work described here is concerned with the application of genetic algorithms (GAs) as optimization tools.

Since their formal introduction in 1975 by Holland [5], genetic algorithms have been applied to a variety of fields—from medicine and engineering to business—to optimize functions which do not lend themselves to optimization by traditional methods, and other applications of GAs include automatic programming and simulation of natural systems. More recently, the study and practical development of the GA by Goldberg [6] and DeJong [7] has resulted in great growth in the application of GAs to optimization problems. As succinctly stated by Goldberg, GAs are "search procedures based on the mechanics of natural selection and natural genetics." Random choice is used as a tool to guide a global search in the space of potential solutions.

GAs differ from traditional optimization and search methods in several respects. Rather than focusing on a single candidate solution (point in design space), genetic algorithms operate on populations of candidate solutions, and the search process favors the reproduction of individuals with better fitness values than those of previous generations (optimal individuals). Whereas calculus-based and gradient (hillclimbing) methods of solution are local in the scope of their search and depend on well-defined gradients in the search space, GAs are useful for dealing with many practical problems containing noisy or discontinuous fitness values. Enumerative searches are also inappropriate for many practical problems. Because they exhaustively examine the entire search space for solutions, they are only efficient for small search spaces, while the global scope of the GA makes it suitable for problems with large search spaces. Thus, GAs not only differ in approach from traditional optimization methods but also offer an alternative method for cases in which traditional methods are inappropriate.

GAs have been applied (or misapplied) to continuous optimization problems, but that is rarely as effective as continuous optimization methods. Evolutionary programming is appropriate for continuous problems; GAs are not, being inherently discrete. The genetic algorithm as a discrete optimization process is distinct from more conventional optimization techniques in four ways:

1. GAs encode designs (feasible points) in a string, and it is this encoding which the GA works with: each individual in a population is an encoding of a possible solution to the discrete optimization problem being analyzed.
2. GAs work simultaneously with a population of designs, not a single design or candidate solution.
3. GAs use only an objective function to evaluate candidate solutions, not derivatives or other auxiliary information.
4. GAs use random change in their search, not (solely) deterministic rules.

The process used by genetic algorithms to evolve solutions to optimization problems is analogous to the natural process of evolution by natural selection. Evolution as a natural process allows complex, highly adapted organisms to develop and thrive in an environment through the processes of genetic change and natural selection. Sexual reproduction (sexual in the sense of occurring between two parent individuals as opposed to one) provides for the preservation of existing genetic information and the creation of new genetic information, and individuals in a population survive based on their fitness in their environment. Fitness is a quality measure of an individual's viability with respect to such criteria in the natural environment as food supply, competition for food and mates, and predation. The genetic information carried by more fit individuals is more likely to be passed on to ensuing generations simply because more fit individuals are more likely to survive to reproduce—Darwinian survival of the fittest.

GAs apply the natural evolutionary processes of evaluation and selection to string representations of the arguments of the function being optimized. Structures (individuals in natural systems) are encoded into one or more strings (chromosomes). These individuals reproduce, and fit individuals persist from from generation to generation, yielding improved designs.

The structure is analogous to the phenotype in natural systems and corresponds to a candidate solution to the optimization problem or a point in the design space, while the string encoding of the arguments to the function being optimized is analogous to the genotype. A decoding from the string representation to the structure is made for the purpose of fitness analysis by the objective function. The objective function yields a quantitative measure of an individual's utility or goodness, to be used as a selection criterion.

For sets of individuals (populations), evolution is simulated by means of reproduction and random genetic changes effected by genetic operators, and survival of the fittest is accomplished by first evaluating each structure's objective function value, and then selecting for reproduction and survival those structures which fit a predetermined selection criterion, biased towards selecting fitter individuals. The search is exploitative: selection is accomplished by analyzing the objective function value with the goal of preserving genetic information which minimizes the objective function. More formally, the aim of the search is to identify an approximation of the global minimum of a real-valued objective function $f\colon M \to E$, by evolving a solution $x^* \in M$ such that $\forall x \in M\colon f(x) \geq f(x^*)$. The process is analogous if the goal is to maximize the objective function.

3. DESIGNING A GENETIC ALGORITHM

The genetic algorithm is a heuristic search process, and its behavior is governed by the following design choices:

1. How shall an individual be represented—what are the values taken on by the genes in the chromosome strings encoding the arguments to the function being optimized?
2. By what means will new individuals be created—what is the mechanism of reproduction?
3. How shall the random genetic processes be accomplished—which genetic operators will be used?
4. By what means will fit individuals in a population be selected for reproduction—what is the mechanism of selection?

The following sections discuss these choices in more detail.

3.1. Representation of Individuals

The genetic information that is operated on by a GA is contained in the chromosomes. A chromosome contains an encoding of the variables of the problem being optimized, and is a finite-length string comprised of elements from a finite alphabet. A gene is a position in the chromosome string, and may take on values from the alphabet; the alphabet is analogous to the set of alleles in a natural system. The nature of the alphabet used to encode the chromosome strings depends on the particular problem being optimized. Goldberg's *principle of meaningful building blocks* and *principle of minimum alphabets* [6] indicate that low-cardinality alphabets should be used—and often variables are encoded using binary or Gray codes. Higher cardinality alphabets are used if the problem requires it, and in fact real-valued genes have been used [4] to encode real variables.

3.2. Reproduction of Individuals

In genetic algorithms, evolution from generation to generation is simulated both by preserving the genetic information contained in the chromosome strings of fit individuals and by altering this information by means of random genetic changes. Both of these goals are effected by genetic operators.

The goal of preserving the genetic information of fit individuals is achieved through crossover. Crossover creates child individuals by crossing over portions of two parent individuals' chromosomes. One or both of the child individuals are retained in the new child population, and the child individuals are required to be unique with respect to the other children and to the parent population. The child population is unique but the crossover operator ensures that the genetic information of the parent population is preserved.

During a one point crossover, two parent individuals are selected at random (with selection biased towards choosing the fittest parents), and their chromosome strings are cut at a randomly determined point. A child individual then receives a chromosome string comprised of the first portion of the first parent's chromosome string and the second portion of the second parent's chromosome string—and so on for crossover operations in which the parent chromosome strings are cut at more than one random point (e.g., two point crossover, uniform crossover). Thus, a unique child individual is created which includes portions of the chromosome strings of both its parents.

In the following example, a two-point crossover is applied to two parent chromosome strings, and the resulting child chromosomes are shown. Genes in the depicted chromosomes take on values from the alphabet $V = \{1\ ,2\ ,3\}$, and 0 is taken to denote an absent—or deleted—gene. The maximum chromosome length is 15, and the number of genes in the chromosome can be less than 15 (e.g., parent chromosome 2 in this example contains 11 genes). The randomly selected crossover points are denoted by the symbol | .

parent chromosome 1 [3 2 3 1|3 3 1 1|3 2 3 1 0 0 0]
parent chromosome 2 [1 1 2 1|3 1 2 2|2 2 1 0 0 0 0]

child chromosome 1 [3 2 3 1|3 1 2 2|3 2 3 1 0 0 0]
child chromosome 2 [1 1 2 1|3 3 1 1|2 2 1 0 0 0 0]

3.3. Introducing Random Genetic Change

The goal of introducing change to the information in the chromosome strings of individuals created by crossover is achieved with the mutation, addition, deletion, and permutation operators. The mutation operator introduces new information into the chromosome string of an individual by randomly altering one or more genes in that string. The following example illustrates a one point mutation carried out on child chromosome 2 from the above crossover operation. A randomly determined gene in the chromosome is changed to take on a new value from V. The mutated gene is indicated with an underscore character _ .

chromosome before mutation [3 2 3 1 3 3 $\underline{1}$ 1 3 2 3 1 0 0 0]
chromosome after mutation [3 2 3 1 3 3 $\underline{2}$ 1 3 2 3 1 0 0 0]

The addition operator randomly adds a gene to the chromosome string. In the following example, a randomly determined gene from V is added at a random point in the chromosome. The randomly selected addition point is denoted by the symbol | . In this example, addition causes the number of actual genes in the chromosome to increase from 12 to 13.

chromosome before addition	[3 2 3 1 3 3 2 1 3|2 3 1 0 0 0]
chromosome after addition	[3 2 3 1 3 3 2 1 3 3 2 3 1 0 0]

The deletion operator randomly deletes a gene from the chromosome string. In the following example, a randomly determined gene, indicated by an underscore character _ , is removed from the chromosome. In this example, deletion causes the number of actual genes in the chromosome to decrease from 13 to 12.

chromosome before deletion	[3 2 3 1 3 $\underline{3}$ 2 1 3 3 2 3 1 0 0]
chromosome after deletion	[3 2 3 1 3 2 1 3 3 2 3 1 0 0 0]

The permutation operator relays information from one part of the chromosome to another by inverting the order of a randomly determined sequence of genes. In the following example, the points at which the permutation operator is applied are indicated by the symbol | .

chromosome before permutation	[3 2 3|1 3 2 1 3 3|2 3 1 0 0 0]
chromosome after permutation	[3 2 3|3 3 1 2 3 1|2 3 1 0 0 0]

The swap operator, like the permutation operator, relays information from one part of the chromosome to the other. The swap operator switches the positions of two randomly determined genes in the chromosome. The swapped genes are indicated with an underscore character _ in the following example.

chromosome before swap	[3 2 $\underline{3}$ 3 3 $\underline{1}$ 2 3 1 2 3 1 0 0 0]
chromosome after swap	[3 2 $\underline{1}$ 3 3 $\underline{3}$ 2 3 1 2 3 1 0 0 0]

3.4. Selection for Reproduction

In GAs, the goal of simulating natural selection is achieved by implementing a selection mechanism. For each generation in the execution of a GA, each individual's chromosome strings are decoded by some decode function, and the decoded individual (the phenotype) is evaluated and given a quality value, or fitness, by the objective function. Individuals are chosen for mating by randomly choosing them from the population, with selection biased towards those individuals with higher relative fitnesses.

Biasing the selection process may be accomplished with, for example, roulette wheel selection (see [6]). The roulette wheel ascribes to each individual a probability of being selected for mating based on its relative position in the population, when the individuals are ranked and sorted according to objective function value. The fitness is part of a simulated roulette wheel, in which the fraction of the roulette wheel, f_i, associated with the i-th best individual in a ranked population of n_d designs is then

$$f_i = \frac{2(n_d + 1 - i)}{n_d^2 + n_d}$$

A uniform random variable determines which portion of the roulette wheel is selected, and the parent individual associated with that portion of the roulette wheel is selected for mating. Thus, the selection of parents for mating and crossover is biased towards those individuals having a more optimal objective function value.

Using a selection scheme instead of simply choosing parents based on their proportional fitness ensures that a highly fit individual does not dominate the population—the likelihood of choosing an individual for mating is based on that individual's relative rank in the population, not on its proportional fitness.

In order to ensure that optimal designs are not discarded during a GA optimization, further selection is done after the child generation has been created. The most fit individual in the parent generation may be retained in the child generation (while discarding the least fit child). This is known as elitist selection. For multimodal problems, several optimal individuals may be retained from generation to generation by using multiple elitist selection. Other selection schemes may be used. Tournament selection, for example, chooses pair of individuals at random from the combined parent and child populations. The more highly fit individual is retained, and the other is discarded. This process is repeated until the new population is full.

3.5. The General Algorithm

Finally, the genetic algorithm may be expressed algorithmically. The implementation of this algorithm is referred to herein as the standard GA (sGA):

```
gen = 0
initialize Population(gen)
decode and evaluate Population(gen)
while not terminated do
    apply crossover to Population(gen) giving Children(gen)
    apply genetic operators to Children(gen)
    decode and evaluate Children(gen)
    Population(gen+1) = select from(Children(gen) ∪ Population(gen))
    gen = gen + 1
    check for termination
end do
```

where `Population` and `Children` are sets of structures, and `gen` is an integer representation of the current generation number. The `terminated` state is determined by some convergence criteria. For example, the GA might run for a fixed number of generations (an epoch), and then terminate, or it might terminate after a certain number of generations have occurred with no improvement in fitness.

4. COMPOSITE LAMINATE STRUCTURE DESIGN AND OPTIMIZATION

4.1. Optimization Methods

Composite materials have received substantial attention as manufacturing materials. This is due to their high stiffness-to-weight and strength-to-weight ratios and their ability to be custom designed to suit the particular environment in which they are used. A composite structure usually consists of one or more laminates. A laminate is comprised of stacks of thin layers of material—plies—where each ply is composed of small diameter fibers of a particular orientation and material type. The plies are held together by a matrix material such as epoxy, which serves to support the fibers and distribute the load amongst them. The strength and stiffness of the fibers is strongest in the direction of their orientation, and weakest in the direction perpendicular to their orientation [5].

The goal of composite laminate design is to find the number of plies, along with the plies' material types and orientations, that yield the best performance for a given set of loading conditions. Additional buckling, manufacturing, or geometry constraints may also be applied to the design. Analysis of composite laminate designs can be computationally expensive, and much of the research effort in this area is concerned with improving the efficiency of the various optimization methods.

Various optimization methods have been applied to finding the optimal stacking sequence for the smallest number of plies which satisfy a given set of design requirements. Using ply orientation angles and the number of plies as design variables, random search has been employed [9], as well as exhaustive search through the entire solution space [10]. Again using ply orientation and thickness as design variables, the problem has also been formulated as a continuous optimization problem [11].

4.2. Genetic Algorithm Optimization

The optimization methods mentioned in the preceding subsection often prove to be impractical for composite material design. Manufacturing limitations limit ply thicknesses and orientation angles to discrete sets of values, so formulation of the composite design problem as a continuous optimization often yields suboptimal or infeasible designs when the design variables are rounded to allowable values. Also, these types of problems often involve nonlinear functions of the design variables, requiring unwieldy approximations and transformation to get tractable linear problems. Finally, composite laminate design problems often admit many distinct globally optimal solutions, instead of one unique global solution.

GAs, as discussed in sections 2.0 and 3.0, are robust tools for discrete optimization problems. They work well with design problems having noisy or nonlinear functions In addition, GAs are global in their scope, and are unlikely to be trapped in local optimal designs. GAs also work with a population of designs, so many distinct optimal or near-optimal designs are found during a GA run. For these reasons, GAs are well suited to the problem of stacking sequence optimization.

4.3. A Fortran 90 GA Module

GAs have been used extensively in the design of composite laminates (e.g., [12]-[16]) and stiffened panels made up of multiple composite laminates ([19],[20]). Many computer programs have been written to implement GAs for specific composite laminate design problems, mostly in FORTRAN 77. The design effort described in this paper has three objectives. First, a Fortran 90 framework is developed for solving composite laminate design optimization problems with GAs. This framework includes a GA module, a package of GA operators, and a package of GA selection schemes. Second, the resulting GA template and package of GA operators are tested on two types of design problems.

Third, the Fortran 90 GA framework is modified to incorporate a parallel distributed migration scheme.

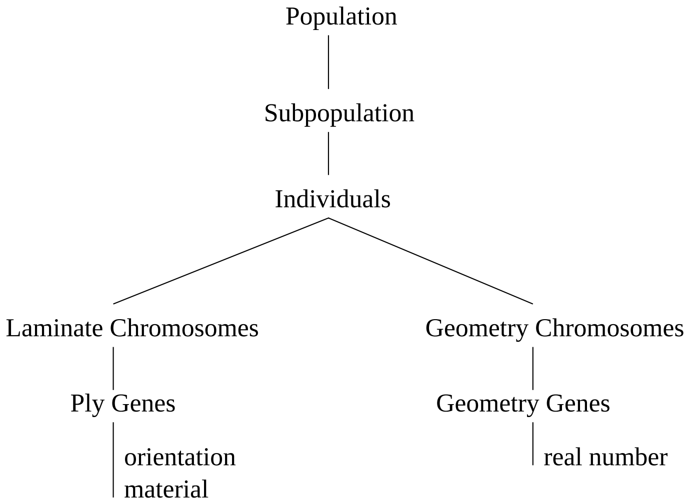

Figure 1—The structure of a population.

The Fortran 90 GA template implements the representation and initialization of a population of designs, as depicted in Figure 1. A population consists of one or more subpopulations, and each subpopulation consists of a set of individual designs. Each individual design in the subpopulation is constructed of zero or more laminate chromosomes and zero or more geometry chromosomes. The laminate chromosomes contain discrete genes encoding the ply material types and orientations, and the geometry chromosomes contain real-valued genes for the optimization of (unencoded) structure geometry variables with an evolutionary strategy. Zero chromosomes are allowed for optimizations in which the design parameters are exclusively discrete (zero geometry chromosomes) or exclusively continuous (zero laminate chromosomes). The makeup of the structure being analyzed—e.g., the number and type of chromosomes, the genetic alphabets, the ranges for the real variables—is parameterized and specific composite laminate structures can be configured through user input.

4.4. A Fortran 90 GA Operators Package

The Fortran 90 GA operators package implements the genetic operators, parent selection, and the determination of the next generation described in section 3.0. The genetic operators include *crossover, mutation, permutation, addition*, and *deletion.* The parent selection scheme used by the package is the roulette wheel. The three methods implemented for determination of the next generation include elitist selection and two types of multiple elitist selection. The GA module includes data structures for continuous variable representation in the geometry genes. Thus, the GA operators package *crossover* includes an evolutionary algorithm style continuous variable crossover. A continuous variable (i.e., a structure geometry variable) is represented by its actual real value, and crossover between two parents is conducted as follows. Given two parents with real valued geometry genes x_1 and x_2, the geometry crossover first creates a randomly distributed $N(\mu, \sigma)$ real number r, where

$$\mu = \frac{x_1 + x_2}{2}, \qquad \sigma = \frac{|x_1 - x_2|}{2},$$

and then enforces physical limits by taking the child value as

$$c = \min\{\max\{r, L\}, U\},$$

where L, U are lower, upper limits on the geometry variable. Note that with probability 0.68 (the probability that a normal variate lies within one standard deviation of the mean μ), c lies between the parent values x_1 and x_2, but that c can also be well outside the segment $[x_1, x_2]$. The continuous variable crossover is essentially how evolutionary algorithms [4] work, where the value of σ itself can be adapted as the evolution proceeds.

The benefit of the GA module and package of operators is that the design of a genetic algorithm for composite material design optimization is simplified. There is no need to develop customized data structures for a particular problem. The GA design reduces to the problem of interfacing the GA module to the analysis routines used with the structure of interest and specifying which genetic operators to apply in the GA design process. Figure 2 depicts the relationship among the program units in a GA optimization using the GA module. The user written main program provides the necessary interface between the GA module, the package of operators designed to work with module data structures, and the specific analysis routines used in the optimization.

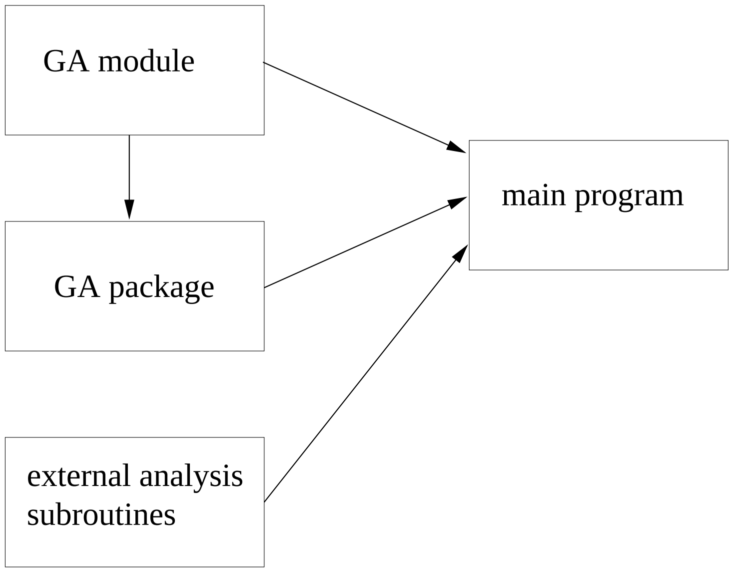

Figure 2—The relationship among program units in a Fortran 90 GA optimizationi.

5. TEST PROBLEMS

Two main programs were developed to interface existing analysis codes to the GA module and package of operators. The analyses chosen for the test programs were previously existing FORTRAN 77 analysis packages. These optimization problems were solved with custom-designed FORTRAN 77 GA codes, and the extensive results establish a good basis for verifying the performance of the Fortran 90 GA module and operators package. This section describes the performance criteria by which the GA analyses are evaluated. Then the two test problems are described. Finally, the results of the test runs are presented. The performance results of this section establish a baseline with which to compare the performance of the migration algorithm presented in the next section.

5.1. GA Performance Analysis

This section introduces the two criteria used for characterizing the performance of the GA during a set of optimization runs: the apparent reliability and the normalized cost per genetic search [16]. The apparent reliability R is a reflection of the reliability of the GA in finding an optimum for a given set of conditions. R is determined by dividing the number of runs in which at least one optimum is found by the total number of runs conducted, N_r .

The normalized cost per genetic search C_n is a reflection of the average number of analyses required to find each unique optimum. C_n is defined by

$$C_n = \frac{N_g S}{A},$$

where N_g is the average number of generations per run over N_r runs, S is the size of a subpopulation, and A is the average number of optima found per run. A is determined by

$$A = \frac{\sum_{j=1}^{N_r} N_o^j}{N_r},$$

where N_o^j is the number of optima found in the jth run. Ideally (R=1.0), $A = 1$ for problems containing a single optimum (e.g., the first test problem) and $A > 1$ for problems containing multiple optima (e.g., the second test problem).

5.2. Test Problem 1 Description: A Single Optimum Design Problem

The analysis code selected for Test Problem 1 analyzes a composite laminate panel measuring 36.0 in. by 30.0 in. The panel is simply supported an all four sides, and can be loaded under any combination of axial and shear loads (N_x, N_y, N_{xy}), as depicted in Figure 3. For simplicity, the panel is assumed to be symmetric about its midplane. This assumption is automatically satisfied because the genetic representation of the laminate only codes one half of the laminate. Further, the panel is required to be balanced. The balance constraint ensures that each ply oriented at $\theta°$ is complemented with another ply oriented at $-\theta°$ throughout the laminate stacking sequence. This constraint is enforced by means of a penalty function in the analysis code. The analysis code determines the load handling capability of a laminate by computing the margin of safety for its critical buckling load and the margin of safety for its critical ply strains. The fitness value returned by the analysis code is a measure of the panel's weight penalized by small or violated safety margins (and the balance constraint penalty). The goal of the GA is to minimize this fitness value. These penalty functions are subtle, and must be chosen carefully so that an infeasible design is not more fit than a near optimal feasible design. Such details are discussed in [17], [18].

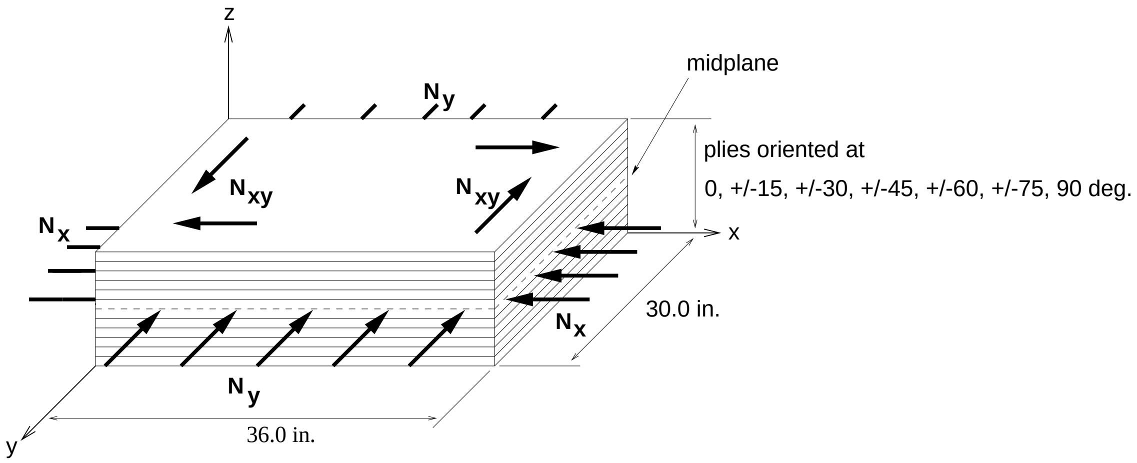

Figure 3—Panel configuration and loading conditions for the two material test problem.

The panel design is encoded in a single chromosome. Each gene in the chromosome corresponds to a ply in the structure. A ply may be composed of either of two materials, Graphite-epoxy (T300/N5208) and Aramid-epoxy (Kevlar 49/CE3305), so the genetic alphabet for the material type genes is $V_m = \{1, 2\}$. A ply may take on any orientation between $-75°$ and $90°$, in increments of $15°$. The balance constraint in the analysis forces plies of the same orientation and material type to take on alternating signs. For instance, the first occurrence of the $\pm 45°$ allele is taken to have the value $+45°$, and the next occurrence is taken to have the value $-45°$. $90°$ and $0°$ plies are not assigned a sign. Thus the ply orientation alphabet V_o uses seven unique alleles to code the twelve ply angles, giving $V_o = \{1, 2, 3, 4, 5, 6, 7\}$ for the set of possible orientations. As an example, consider the following chromosome strings, where each (orientation, material) pair is a gene:

orientation alleles	[1 1 3 3 3 3 4 4 7 7 0 0 0 0]
material type alleles	[1 1 1 1 1 2 2 2 2 2 0 0 0 0]

When this genotype is decoded by the analysis routine, the design (phenotype) is the following:

$$\text{decoded design} \qquad [0_2^{(1)} / \pm 30_3^{(1)} / \pm 30_1^{(2)} / \pm 45_2^{(2)} / 90_2^{(2)}]_s$$

where $\theta_n^{(m)}$ represents n plies of material type m at orientation θ. The subscript $_s$ indicates that the decoded design is taken to be one half the full set of plies, with the full design symmetric about the midplane, corresponding to the right end of the string.

This particular design problem contains a single unique global optimum solution amidst many near-optimum solutions. In order to locate this optimum (force convergence) while also enabling the most efficient use of the GA, an elitist selection scheme is used for determining the next generation. The elitist selection scheme works as follows. For each iteration of the GA loop, a child population is generated. The child population is the same size as the parent population. The elitist selection algorithm (EL) determines the most fit parent design and retains it in the child generation, making room for the elite parent individual by discarding the least fit child individual. Thus, the most fit design is always retained, ensuring that each new generation's most fit individual is at least as fit as the previous generation's. In addition, all but one of each new generation's designs are unique results of the application of the genetic operators, i.e., duplicate children are not permitted, ensuring diversity in the possible solutions examined.

The input parameters to the GA module for Test Problem 1 for all GA runs in this paper are as follows: probability of crossover $p_c = 1.0$, probability of material mutation $p_{mm} = 0.15$, probability of orientation mutation $p_{mo} = 0.15$, probability of ply addition $p_a = 0.05$, probability of ply deletion $p_d = 0.10$, probability of ply swap $p_s = 0.75$. The results for 64 separately evolved subpopulations using the standard GA (sGA) are enumerated in Table 1. For these runs, the subpopulation size is set to 60, and the termination criterion for a run is 400 consecutive generations with no improvement in the best fitness.

5.3. Test Problem 2 Description: A Multiple Optimum Design Problem

The analysis code selected for Test Problem 2 analyzes a composite laminate panel measuring 20.0 in. by 10.0 in. The panel is simply supported an all four sides, and can be loaded under any combination of axial loads (N_x, N_y), as depicted in Figure 4. As with Test Problem 1, the panel is assumed to be balanced and symmetric about its midplane. The analysis package used with this test problem determines the buckling load of a given design by determining the critical buckling load factor. The goal of the GA is to maximize the failure load of the laminate. In Test Problem 1, the number of plies is variable; here the number of plies is fixed at 20.

The panel design is encoded in a single chromosome. Each gene in the chromosome corresponds to a pair of plies in the structure. The laminate is composed of one material (graphite epoxy) and each ply pair may take on an orientation of 0°, ±45°, or 90°. Because the plies occur in pairs, a gene coding for 0° decodes to two contiguous 0° plies, the ±45° gene decodes to two contiguous plies of +45° and −45° orientation, and the 90° gene decodes to two contiguous 90° plies. The laminate is assumed to be symmetric about its midplane, satisfying the symmetry constraint, and the balance constraint is satisfied by using contiguous two-ply stacks. Thus, the genotype representation of the laminate represents one fourth of the decoded phenotype.

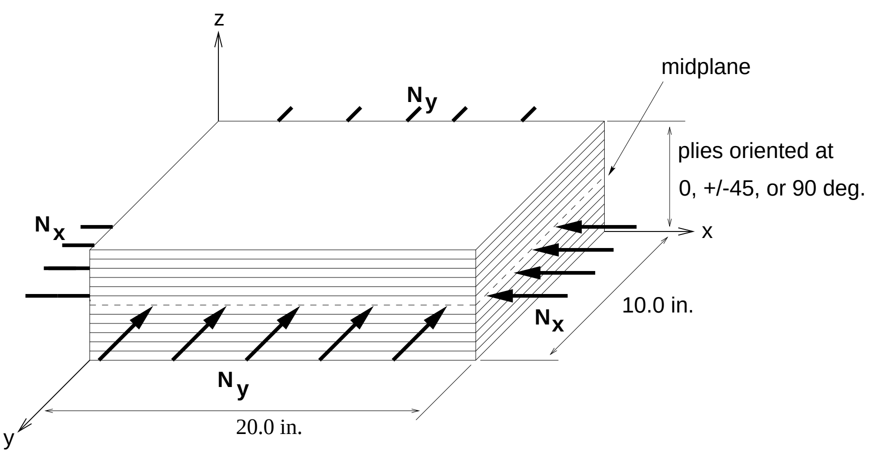

Figure 4—Panel configuration and loading conditions for the multimodal test problem.

The input parameters to the GA module for Test Problem 2 for all GA runs in this paper are as follows: probability of crossover $p_c = 1.0$, probability of material mutation $p_{mm} = 0.0$

(only one material), probability of orientation mutation $p_{mo} = 0.02$, probability of ply addition $p_a = 0.0$, probability of ply deletion $p_d = 0.0$, probability of ply swap $p_s = 0.05$, probability of ply permutation $p_p = 0.02$. The results for 64 separately evolved subpopulations using the standard GA (sGA) are enumerated in Table 1. For these runs, the subpopulation size is set to 60, and the termination criterion for a run is 100 consecutive generations with no improvement in the best fitness.

The number consecutive runs to convergence for this problem is smaller than for Test Problem 1 (100 vs. 400), because the number of possible designs is significantly smaller for Test Problem 2. The number of possible designs is smaller because of the smaller orientation alphabet (three alleles here vs. seven for Test Problem 1) and the use of a single material. The results for 64 separate optimization runs using the standard GA (sGA) are enumerated in Table 1.

Whereas Test Problem 1 contains a single unique globally optimum solution, this particular design problem contains a group of globally optimum designs. In order to make efficient se of the GA, a variable elitist selection scheme is used for determining the next generation. The multiple elitist selection algorithm is designed to retain more than one individual from the parent generation in the child generation. This strategy represents a performance tradeoff for the GA: exploitation of highly fit designs occurs by retaining elite designs from generation to generation throughout the GA run, but exploration of further candidate designs is compromised as the number of retained individuals is increased. Two different multiple elitist schemes are applied to test problem 2. These strategies are inspired by the (μ, λ) and $(\mu + \lambda)$ strategies developed by Bäck ([21], [22]).

The first multiple elitist selection strategy (ME1) works as follows: for each iteration of the GA loop, a child population is generated equal in size to the parent population. The children are required to be distinct from each other and from all the parents. ME1 determines the n_k most fit designs in the parent population, where n_k is specified at run time. The n_k most fit parent individuals replace the n_k least fit individuals of the child generation. The n_k least fit members of the child generation are discarded to make room for the retained parents. This strategy is similar to the elitist strategy used in Test Problem 1, and, in fact, ME1 reduces to EL for $n_k = 1$.

The second multiple elitist selection strategy (ME2) works as follows: for each iteration of the GA loop, a child population is generated equal in size to the parent population. The children are required to be distinct from each other and from all the parents. The parent and child populations are combined, and the n_k most fit individuals from the combined population are determined. These n_k individuals are retained in the child generation, which is completed using the most fit remaining children, yielding a child population equal in size to the parent population.

5.4. GA Performance Results for Test Problems 1 and 2

For both test problems, the Fortran 90 GA implementation yielded optimum designs identical to those found by the original custom designed FORTRAN 77 GA codes (a single optimum for Test Problem 1, and 6 global optima for Test Problem 2). The globally optimum designs for the two test problems are enumerated in Appendix A. Also, see [23] for a discussion of language performance tradeoffs between the custom FORTRAN 77 codes and the Fortran 90 template. The test runs described in sections 5.2 and 5.3 establish a baseline for comparison with the migration algorithm presented in the next section .

Table 1
sGA performance results— independently evolving subpopulations.

Test Problem	Selection	n_k	C_n	R	A
1	EL	*	$65,457$	0.78	0.78
2	EL	*	$9,182$	1.0	1.00
2	ME1	5	$2,167$	1.0	4.16
2	ME1	15	$2,812$	1.0	3.50
2	ME1	30	$3,800$	1.0	2.58
2	ME2	5	$2,320$	1.0	4.15
2	ME2	15	$2,846$	1.0	3.53
2	ME2	30	$3,840$	1.0	2.76

* n_k is not applicable for EL selection.

The normalized cost and reliability for these runs are reported in Table 1. Test Problem 1 converges with a reliability of 0.78 for EL selection. All 64 subpopulations converged (400 consecutive generations without an improvement in the best fitness), but not all converged to the best known optimum. All 64 subpopulations did, however, converge to within 5% of the best known optimum fitness. Generally, a reliability of 0.80–0.90 is considered acceptable, so one of the goals of the migration algorithm is to improve R at equal or less cost.

All selection schemes for Test Problem 2 converge with a reliability of 1.0, and with a lower normalized cost per optimum than Test Problem 1. The lower cost is attributable to the smaller search space and the smaller genetic alphabet of Test Problem 2—there are not as many candidate solutions to consider. A large improvement in C_n is seen when using ME1/ME2 selection rather than EL selection. The ME selection routines exploit previously found highly fit designs by retaining them from generation to generation. Increasing the number of retained designs serves to increase the number of highly fit individuals involved in the reproduction process. This is, however, at the expense of exploration. Increasing the number of retained designs decreases the space available in the population for consideration of new designs. This is evidenced by the increasing normalized cost of both ME schemes as n_k is increased.

In the next section, a distributed parallel migration algorithm is presented, and the effect of the migration scheme on C_n and R for the Test Problems 1 and 2 is explored.

6. PARALLEL IMPLEMENTATION

As parallel computers become more commonplace in scientific computing, it becomes more feasible to harness their power for use with genetic algorithms. The GA is inherently parallel [24]. Genetic operations can be applied to individuals in a population in parallel, a population may be partitioned into separately evolving subpopulations, or independent runs—such as those made in Section 5—can run on different processors. A GA running in parallel can be exploited further as an optimization tool by implementing communications between processing elements. Indeed, Gordon and Whitley [26] report that the performance of parallel genetic algorithms is superior to standard GAs in function optimization, even without taking parallel hardware into account.

The migration model takes the idea of separately evolving subpopulations and extends it by adding a means of selectively sharing genetic information between them. Migration may occur in a variety of ways. Each processing element in a parallel GA may contain an independently evolving standard GA which periodically migrates fit individuals to other processors (as in [25]), or the GA itself may be parallelized. In the latter case, each individual in a population is itself a single process, and migration is characterized as a diffusion process as individuals reproduce within a local neighborhood (see, for example [27]).

The implementation of a distributed migration algorithm should ensure that the performance of the algorithm is not constrained by the number of processors involved or by the communication between those processors. The mechanisms of communication, load balancing, and termination detection all determine the distributed algorithm's parallel performance. The goal of the migration algorithm presented here is to extend the sGA presented in Section 5 to implement a migration scheme on a distributed memory parallel machine. The present section describes the implementation of the distributed migration algorithm incorporated into the Fortran 90 GA framework (the distributed GA is referred to as dGA). All parallel code used in the dGA extension is implemented using the MPI 1.2 (Message Passing Interface) standard for FORTRAN. Next, experiments are conducted to compare the performance of the dGA with that of the sGA baseline established in Section 5. Section 7 quantifies the performance of the migration algorithm and the performance of the parallel algorithm.

6.1. Migration

The migration algorithm partitions a population of designs into a set of subpopulations and, at specified intervals, shares information between these subpopulations. Tanese [29] introduces the parameters associated with the migration algorithm: the migration interval and the migration rate. The migration interval is the number of generations between each migration, and the migration rate is the number of individuals selected for migration.

For the migration algorithm described in this paper, migrating subpopulations are arranged in a ring topology. Migration occurs between directionally adjacent subpopulations, as depicted in Figure 5. The migration interval is incorporated into the distributed algorithm as a probability p_m, and the migration rate is incorporated as a maximum value n_m.

For each subpopulation in the distributed GA, migration is accomplished as follows. At the end of a generation, a uniformly distributed random number y is generated. If $y < p_m$, migration is initiated. During migration, a uniform random number determines the number of individuals n_s between 1 and n_m to send. The best n_s individuals in the subpopulation are sent to the nearest neighbor in the ring. Whether or not emigrants are sent to the nearest neighbor, the subpopulation then checks to see if immigrants are arriving from its neighbor. If immigrants are arriving, they are received into the subpopulation and replace the n_s least fit individuals.

Migration intervals are typically specified as a fixed number of generations, known as an epoch. The problem with using a fixed epoch value is that migration is globally synchronized across all subpopulations. Using a random interval allows the subpopulations to evolve asynchronously. The subpopulations are not required to migrate in lockstep at a prescribed interval. An additional advantage conferred by asynchronous migration is that communications between the subpopulations are spread out in time more than if migration were synchronized. This makes more efficient use of communications bandwidth and allows quickly converging subpopulations to finish early, freeing up their processors to do more work. Some control over average length of an epoch exists. For

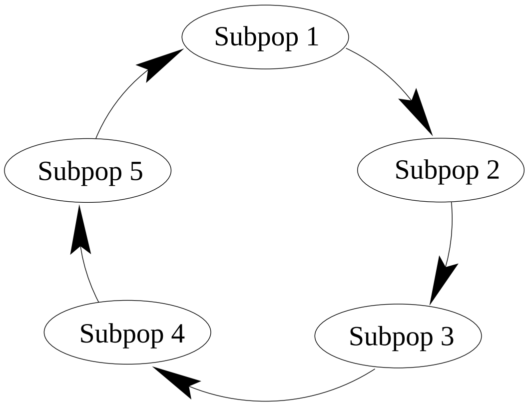

Figure 5—Ring topology for migration among subpopulations.

example, the expected value of the epoch length for $p_m = 0.01$ is $1/p_m = 100$ generations—it is expected that, on average, migration occurs once every 100 generations for $p_m = 0.01$.

As with multiple elitist selection, migration represents a tradeoff between exploration of new designs and exploitation of highly fit designs which have already been found. The physical relationship between subpopulations imposed by the topology of the distributed system has an effect on this tradeoff as well. The ring topology used for the dGA described in this paper ensures local communications between subpopulations. The benefit of this design is that migration occurs locally between adjacent populations on the ring. This yields local exploitation of fit designs, while globally the separate subpopulations are free to explore different types of designs independently.

For composite laminate design, the analysis of individual designs is often the most expensive part of the GA solution. Although it is reasonable to expect that larger subpopulations of individuals might perform better than smaller subpopulations, increasing the subpopulation size P results in an increase in C_n. The goal of the dGA is to achieve better GA performance than the (serial) sGA for a given P, with a concomitant improvement in parallel performance over a parallel sGA. Experiments designed to test the performance of the dGA are described in Section 7.

6.2. An Extension to the sGA

The standard GA is extended to yield the distributed GA with migration (dGA). Each processing element in the dGA algorithm performs the following code:

```
gen = 0
initialize Population(gen)
decode and evaluate Population(gen)
while not terminated do
    apply crossover to Population(gen) giving Children(gen)
    apply genetic operators to Children(gen)
    decode and evaluate Children(gen)
    Population(gen+1) = select from(Children(gen) ∪ Population(gen+1))

    begin migration
      if migration appropriate
         choose emigrants
         send emigrants
      end if
      if immigrants available
         receive immigrants
         Population(gen+1) = select from(immigrants ∪ Population(gen+1))
      end if
   end migration

   gen = gen + 1
   check for termination
end do
```

6.3. Dynamic Load Balancing and Distributed Control

Individual subpopulations in a GA run do not involve the same amount of work as other subpopulations in the same run. Some subpopulations will converge early, and others will converge more sluggishly, yielding a much higher normalized cost. The result is considerable variation in the number of generations to convergence from subpopulation to subpopulation. A simple static load balancing scheme for a parallel GA optimization using p processors and S subpopulations might assign S/p subpopulations to each processor, and then allow each processor to evolve those subpopulations with no further load balancing. However, due to the variation in the number of generations to convergence, a load imbalance will inevitably occur with this scheme, as some processors finish their work well before others finish.

The dGA described in this paper uses a dynamic load balancing strategy which redistributes work to processors which finish their work early and become idle. The dynamic load balancing strategy used with the dGA initially assigns S/p subpopulations to each processor. If a processor runs out of work, it queries the other processors to see if there is work which may be shared (greater than one remaining subpopulation on the queried processor). This query is conducted in an asynchronous fashion, starting at the nearest neighbor on the ring. If there is work to be shared, the two processors divide the remaining subpopulations between themselves, and the GA algorithm is resumed. If there is no work to be shared, the querying processor is terminated.

Sharing of work between processors is only done at the request of the receiving process, and only the receiving process and the requested process are involved in this communication. Thus, there is no global control of the computation; control is distributed. In order for distributed control to be useful, a means of termination detection is required. This is discussed next.

6.4. Termination Detection

It is a simple enough matter to detect termination on a single processor. However, when using multiple processors in a distributed system, termination detection becomes more difficult because there is no knowledge of the global state of the system. The ring topology selected for implementing the dGA is also suitable for a token passing algorithm for detecting global termination. The token-passing algorithm used here operates as follows. When p_0 (the first processor in the ring) is finished evolving all its subpopulations, and no other processors can share work with it, a token is passed to the next processor in the ring. Each processor passes the token along under the same conditions. At the point when p_0 receives the token again from the last processor in the ring, it is assured that all processors in the ring have indicated a terminated state, and the global computation is thus complete.

7. PARALLEL PERFORMANCE

In order to compare the performance of the distributed migration algorithm (dGA) with that of the standard GA (sGA), two types of experiments were performed: a set of experiments to examine the reliability and cost performance of the migration algorithm relative to the sGA, and a set of experiments to examine the timing performance of both the dynamic load balancing algorithm and the migration algorithm. All of the test runs described in this section were conducted on a 128-parallel-processor Cray T3E-900.

7.1. Measuring the Effects of Migration

In the first set of experiments, the dGA is compared directly with the sGA baseline established in Section 5. The sGA baseline was established using 64 subpopulations (each of size 60) evolving in isolation. The dGA comparison is made using 32 subpopulations (each of size 60) evolving with migration. For both test problems, n_m and p_m are varied, and C_n and R are reported.

Next, a comparison is made between the dGA and a set of sGA runs using varying subpopulation sizes. The input parameters to the GA are the same as those used for establishing the sGA baseline results, and C_n is reported. These results are compared to the results of the first experiment to compare the effect of varying subpopulation size to that of migration. The results for both these experiments are enumerated in Tables 2–4 and discussed in the next section.

7.2. Measuring Parallel Execution Time

Whereas the migration algorithm seeks to improve upon the normalized cost and reliability of the sGA, it is important also to ensure that unacceptable time overhead is not introduced by migration or the dynamic load balancing algorithm. An important measure of the quality of a parallel algorithm is its parallelizability [30]. The parallelizability is the ratio between the time taken by a parallel computer executing a parallel algorithm on one processor and the time taken by the same parallel computer executing the same parallel algorithm on p processors. Because migration and the number of simultaneously migrating subpopulations (the number of processing elements used in the computation) both play a role in what C_n and R will ultimately be for a set of GA runs, it is difficult, if not impossible, to enforce a constant amount of work across all processing elements as the number of processing elements is varied. The parallelizability measurement, to be meaningful, relies on the amount of (useful) work being constant as the number of processors is varied. Thus, parallelizability is not sufficient for characterizing the performance of the dGA, because the dGA is nondeterministic in its parallel behavior. Other performance measures are required.

The second set of experiments characterizing the dGA are intended to examine the time effects of the migration algorithm and the dynamic load balancing algorithm. First, because a direct measurement of parallelizability is not meaningful, an effective parallelizability measurement is considered. For measuring the effective parallelizability, the conditions of a GA optimization run are altered, by changing the convergence criteria for a subpopulation. Instead of declaring convergence after a certain number of generations transpire without an improvement in the fitness of the best individual in the subpopulation, a simple constant number of generations is used. Each subpopulation evolves over a constant number of generations. Next, the amount of work done in the kth subpopulation is fixed, by associating a specific random number seed with k, independent of the number of processors used in the run. Timing runs are made in which the number of processors is varied from 1–32, with the total number of subpopulations (and thus the total amount of work performed) kept constant. The effective parallelizability observed under these conditions is linear—there is no degradation in the time performance of the dGA as the number of processors is increased.

The linear effective parallelizability under the contrived test conditions described in the preceding paragraph is encouraging, but it is reasonable to expect that the migration algorithm itself has a run time cost associated with it. In order to examine this assertion, a single processor is used to conduct a GA optimization on 16 subpopulations, again using the new, fixed convergence criteria and normalized work amount in the kth subpopulation. The probability of migration p_m is varied, and time to termination is recorded. The results of this measurement are listed in Table 5.

7.3. Performance of the Migration Algorithm

Comparing Table 1 with Tables 2 and 3, it is clear that the migration algorithm yields an improvement in normalized cost and in reliability.

For Test Problem 1 (Table 2), an ideal reliability (R=1.0) is achieved, with a concomitant decrease in the normalized cost per optimum found. R is 1.0 at $p_m = 0.05$ and $n_m = 5$, and these values are taken to be the ideal migration parameters for Test Problem 1. As n_m and p_m decrease, C_n and R both approach the values found using the sGA (Table 1).

For Test Problem 2 (Table 3), the trend of decreasing normalized cost is observed as well. It is interesting to note that the reliability actually falls below 1.0 for small values of p_m, indicating

Table 2

dGA performance results, Test Problem 1 (EL selection).

n_m	p_m	C_n	$R(=A)$
5	0.05	46,800	1.0
5	0.02	51,180	1.0
5	0.01	58,560	0.91
5	0.001	72,540	0.82
1	0.05	54,300	1.0
1	0.02	55,500	1.0
1	0.01	70,260	0.90
1	0.001	60,900	0.79

Table 3

dGA performance results, Test Problem 2 (ME1 selection).

n_m	p_m	C_n	R	A
5	0.05	1,670	1.0	5.22
5	0.02	1,844	1.0	4.53
5	0.01	1,980	1.0	3.97
5	0.001	2,175	0.96	3.47
1	0.05	1,540	1.0	5.16
1	0.02	1,721	1.0	4.72
1	0.01	2,032	1.0	4.31
1	0.001	2,194	0.94	3.38

Table 4

sGA performance results for varying subpopulation size P.

	Test Prob 1 (EL)			Test Prob 2 (ME1)		
P	C_n	N_g	A	C_n	N_g	A
960	600,960	626	1.0	20,352	106	6.0
240	172,080	717	1.0	6,840	114	4.0
60	65,457	1091	1.0	2,167	147	4.0
15	*23,190	1546	0.0	1,232	231	2.0

*optimum not found

Table 5

Time in seconds, for varying migration probability
16 subpopulations with migration, $n_m = 5$.

	time	
p_m	Test Prob 1 (EL)	Test Prob 2 (ME1)
0.0	98.31	156.12
0.01	98.41	158.78
0.02	99.25	164.23
0.05	101.01	170.77
0.1	103.09	179.70
0.2	106.83	187.68
0.5	112.31	210.87

that not all subpopulations converged to the best known optimum. These cases contrast with the (sGA) results in Table 1, in which $R = 1.0$. In the case of $R < 1.0$, a small migration probability introduces highly fit but suboptimal designs into a subpopulation, and this subpopulation converges prematurely. Here, premature convergence occurs when the convergence criterion is met (100 consecutive generations without an improvement in the best fitness in the subpopulation) before additional migration or a local change in the best fitness can occur to improve this suboptimal best design. However, as p_m is increased, not only does the reliability improve to 1.0, but the average number A of globally optimal designs found per subpopulation increases as well. This improvement in reliability correlates with a decrease in the normalized cost and an increase in the average number of optimal designs found per subpopulation. i.e., by all measures *higher migration rates are better.*

The intent of the second experiment is to compare the performance of the dGA to that of the sGA, using varying subpopulation size P. From Table 4, varying P has a definite effect on the performance of the algorithm. Two trends are apparent with varying subpopulation size.

The first trend in Table 4 is increasing normalized cost C_n as subpopulation size increases. As C_n increases, the number of generations to convergence N_g decreases, but this decrease is more than offset by C_n. Increasing the subpopulation size is prohibitively expensive in terms of the cost measure used for these test problems. As observed earlier, C_n is an accurate measure of the true cost of a GA run, because the cost in time of evaluating an individual is high relative to the cost of the GA itself. The reason the increase in cost outweighs the decrease in number of generations to convergence is that as the number of individuals in a subpopulation increases, the likelihood of highly ranked individuals being selected for reproduction decreases relative to the rest of the members in the subpopulation. In terms of the roulette wheel selection function, the proportion of the wheel allocated to highly fit individuals decreases as more individuals are added, resulting in a decreased probability of selecting the best individuals in the current generation for reproduction.

The second trend in Table 4 is decreasing normalized cost C_n as subpopulation size decreases. This apparent improvement in results is misleading unless examined in light of the number of optima found A in each GA run. The cost does, in fact decrease, but as C_n decreases, so does the ability of the GA to find optimum solutions. For the smallest subpopulation size ($P = 15$), the GA converges prematurely for Test Problem 1, and an optimum is never found. The decreased cost is at the expense of reliability (no optimum found yields $R = 0.0$) . Contrast this with the higher cost of the migration algorithm, but with a reliability of $R = 1.0$—all subpopulations in the migration run converge to the best known optimum for Test Problem 1. A similar result occurs with Test Problem 2. Although C_n decreases with decreasing subpopulation size, the number of optimum solutions found also decreases. Again, comparing the case of $P = 15$ with the migration algorithm results, we get a higher cost using the migration algorithm, but $A = 5.22$ with migration versus $A = 2.0$ at $P = 15$, without migration.

From Table 5, the migration algorithm has an effect on execution time, compared to the case running without migration ($p_m = 0.00$). The increase in time is essentially linear as p_m is varied. At the points determined to be "best," $p_m = 0.05$ and $n_m = 5$, the additional time is small—a 2.8% increase over $p_m = 0.00$ for Test Problem 1, and an 11.0% increase over $p_m = 0.00$ for Test Problem 2. These values are small when compared to the effect of the migration algorithm on normalized cost. The diminished normalized cost brought about by use of the migration algorithm more than offsets these time penalties—a 28.5% reduction (from $C_n = 65,457$ to $C_n = 46,800$) for Test Problem 1 and a 22.9% reduction (from $C_n = 2,167$ to $C_n = 1,670$) for Test Problem 2.

Table 6

Execution time for dynamic (t_d) and static (t_s) load balancing, $n_m = 5$, $p_m = 0.05$.

S	t_d	t_s	*% difference
1024	2944	2983	1.3%
512	1791	1822	1.7%
256	920	970	5.2%
128	480	516	7.0%
64	247	281	12.1%
32	142	153	7.1%
16	96	101	4.9%
8	51	51	0.0%

* $(t_s - t_d)/t_s \times 100\%$

7.4. Testing the Dynamic Load Balancing Algorithm

In order to analyze the performance of the dynamic load balancing algorithm independent of the migration algorithm, migration is turned off ($p_m = 0.00$), and the original convergence criterion (a certain number of generations without a change in the best fitness) is turned back on. Recall that migration, *per se*, has an effect on normalized cost—and thus execution time—and this effect is dependent upon the number of processors used in the computation. Turning off migration for analysis of the load balancing algorithm ensures that timing measurements indicate the effects of load balancing, rather than the effects of migration. The "repeated generations" convergence criterion is the criterion used for making the original baseline measurements (using the sGA) in Section 5. The consequence of using this convergence criterion is that there is much variation in the number of generations to convergence from subpopulation to subpopulation. The dynamic load balancing algorithm compensates for this variation by reassigning subpopulations on busy processors to idle processors. To observe the effect on execution time of this algorithm, the number of subpopulations S is varied on a fixed number of processors $p = 8$, for Test Problem 2 with ME1 selection and the same parameters given in Section 5. For each run in this experiment, execution times for both the dynamic load balancing algorithm t_d and a static load balancing algorithm t_s are recorded and enumerated in Table 6.

7.5. Dynamic Load Balancing Results

From Table 6, the dynamic load balancing algorithm has some effect on diminishing the execution time for a set of S subpopulations on $p = 8$ processors. At $S = p$ the execution times are identical for both the static and dynamic algorithms, because when a processor runs out of work, there is no work left to be shared by the other processors in the computation—each other processor is either completing its single subpopulation or is finished. As S increases, the number of subpopulations originally allocated to each processor S/p also increases. For small values of $S/p > 1$, the likelihood increases that there will be some sharing of work. The amount of work shared depends on the amount of variation in the number of generations to convergence, which in turn depends on the type of optimization problem being solved. For the test problems addressed in this paper, the ratio between the largest and smallest number of generations to convergence is approximately 3.0. For some problems, this ratio can be as high as 10.0. The amount of load balancing that will be required will vary with this ratio. At the same time, as S/p increases more, the size of the load imbalance decreases, because the average number of generations to convergence for each processor approaches the average on the other processors. Thus, for this algorithm there are two factors contributing to the extent to which load balancing will affect the execution speed of the computation: the factor S/p, and the amount of variation in the total number of generations to convergence. Regardless of these parameters, the least improvement is at $S = p$—where dynamic load balancing is only as good as static load balancing—and at high values of S/p, where variations in the number of generations to convergence for individual subpopulations are offset by the sheer number of subpopulations.

8. CONCLUSION

The distributed genetic algorithm with migration (dGA) has been applied to two types of composite laminate design problems, and its performance compared to that of the standard GA (sGA). The migration algorithm itself results in diminished normalized cost C_n per optima found and improved reliability R, for both of the test problems examined here. Nondeterministic algorithms coupled with dynamic load balancing and distributed control result in variable parallel workloads, making parallel performance evaluation complicated. The experiments designed to characterize the parallel performance of the dynamic load balancing algorithm and the distributed control algorithm with termination detection demonstrate linear effective parallelizability for constant workload. Finally, use of the migration algorithm incurs a small time penalty, which is outweighed by the diminished normalized cost.

Positive results have been achieved using the ring communication topology. Future work with the dGA might include an exploration of the effect of different communication topologies, and the application of both sGA and dGA packages to other types of composite laminate design problems. Such a study might also include a deeper exploration of the tradeoffs associated with using higher migration rates, such as diminished diversity and increased execution time.

ACKNOWLEDGMENTS

This work was supported in part by Air Force Office of Scientific Research grant F49620-96-1-010, National Science Foundation grant DMS-962596, and National Aeronautics and Space Administration grant NAG-2-118.

References

[1] J. M. Bremermann, Optimization through evolution and recombination, *Self-Organizing Systems* **7** (1958), 282–287.

[2] R. M. Friedberg, A learning machine: Part I, *IBM J. of Research and Development* **1** (1958), 2–13.

[3] R. M. Friedberg, A learning machine: Part II, *IBM J. of Research and Development* **7** (1959), 282–287.

[4] T. Bäck, *Evolutionary Algorithms in Theory and Practice* , Oxford University Press, NY, 1996.

[5] J. H. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, MI, 1976.

[6] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning* , Addison-Wesley, Reading, MA, 1989.

[7] K. A. De Jong, *An analysis of the behavior of a class of genetic adaptive systems*, PhD. Dissertation, Univ. of Michigan, Ann Arbor, MI, 1975.

[8] R. T. Haftka, *Elements of Structural Optimization*, Kluwer Academic Publishers, Boston, 1992.

[9] R. Foye, Advanced Design for Advanced Composite Airframes, *Air Force Materials Laboratory,* Wright-Patterson Air Force Base, Ohio, AFML-TR-69-251 (1969).

[10] M. E. Waddoups, Structural Airframe Application of Advanced Composite Materials—Analytical Methods, *Air Force Materials Laboratory,* Wright-Patterson Air Force Base, Ohio, AFML-TR-69-101 (1969).

[11] Z. Gürdal and R. T. Haftka, Optimization of Composite Laminates, Presented at the *NATO Advanced Study Institute on Optimization of Large Structural Systems* (1991), Berchtesgaden, Germany.

[12] R. Le Riche and R. T. Haftka, Optimization of Laminate Stacking Sequence for Buckling Load Maximization by Genetic Algorithm, *AIAA Journal* **31** (1993), 951–956.

[13] R. Le Riche and R. T. Haftka, Improved Genetic Algorithm for Minimum Thickness Composite Laminate Design, Proceedings of International Conf. on Composite Engineering, (1994), New Orleans, LA.

[14] N. Kogiso, L. T. Watson, Z. Gürdal, R. T. Haftka, and S. Nagendra, Design of Composite Laminates by a Genetic Algorithm with Memory, *Mechanics of Composite Materials and Structures* **1** (1994), 95–117.

[15] N. Kogiso, L. T. Watson, Z. Gürdal, and R. T. Haftka, Genetic algorithms with local improvement for composite laminate design, *Structural Optim.* **7** (1994), 207–218.

[16] G. A. Soremekun, *Genetic Algorithms for Composite Laminate Design and Optimization*, MS Thesis, Department of Engineering Mechanics, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1997.

[17] S. Nagendra, D. Jestin, Z. Gürdal, R. T. Haftka, and L. T. Watson, Improved genetic algorithms for the design of stiffened composite panels, *Comput. & Structures* **58** (1996), 543–555.

[18] S. Nagendra, S., Z. Gürdal, R. T. Haftka, and L. T. Watson, "Derivative based approximation for predicting the effect of changes in laminate stacking sequence", *Structural Optim.* **11** (1996), 235–243.

[19] S. Nagendra, R. T. Haftka, and Z. Gürdal, Design of Blade Stiffened Composite Panels by a Genetic Algorithm Approach, *Proceedings of the 34th AIAA/ASME/AHS SDM Conf.*, La Jolla, CA, (1993), 2418–2436.

[20] S. Nagendra, R. T. Haftka, and Z. Gürdal, PASCO-GA : A Genetic Algorithm based Design Procedure For Stiffened Composite Panels under Stability and Strain Constraints, *Tenth DOD/NASA/FAA Conf. on Fibrous Composites in Structural Design*, Hilton Head, SC, (1993).

[21] T. Bäck and F. Hoffmeister, Extended Selection Mechanisms in Genetic Algorithms, *Proceedings of the 4th International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, (1991), 92–99.

[22] T. Bäck, Selective Pressure in Evolutionary Algorithms: A Characterization of Selection Mechanisms, *Proceedings of the First International IEEE Conference on Evolutionary Computation*, IEEE Press, NY, (1994), 57–62.

[23] M. T. McMahon, L. T. Watson, G. A. Soremekun, Z. Gürdal, and R. T. Haftka, A Fortran 90 Genetic Algorithm Module for Composite Laminate Structure Design, *Engineering with Computers* (1998).

[24] J. J. Grefenstette and J. E. Baker, How genetic algorithms work: A critical look at implicit parallelism, *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, Morgan Kaufmann, San Mateo, CA, (1989), 20–27.

[25] J. P. Cohoon, S. U. Hedge, W. N. Martin, and D. Richards, Punctuated equilibria: A parallel genetic algorithm, *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, Morgan Kaufmann, San Mateo, CA, (1987), 750–755.

[26] V. S. Gordon and D. Whitley, Serial and Parallel Genetic Algorithms as Function Optimizers, *Proceedings of the Fifth International Conference on Genetic Algorithms and Their Applications*, Morgan Kaufmann, San Mateo, CA, (1993), 177–183.

[27] M. Gorges-Schleuter, Asparagos: an asynchronous parallel genetic optimization strategy, *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, Morgan Kaufmann, San Mateo, CA, (1987), 422–427.

[28] F. Hoffmeister, Scalable Parallelism by Evolutionary Algorithms, *Lecture Notes in Economics and Mathematical Systems* **367** (1991), Springer Verlag, Berlin, 177–198.

[29] R. Tanese, Distributed Genetic Algorithms, *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, Morgan Kaufmann, San Mateo, CA, (1989), 434–439.

[30] M. J. Quinn, *Parallel Computing, Theory and Practice*, McGraw-Hill, NY, 1994.

Appendix A—Optimum designs for Test Problems 1 & 2.

The single globally optimum design for Test Problem 1.

Design Rank	Stacking Sequence	Weight	Fitness
1	$\pm45_6^{(2)}/90_1^{(1)}/\pm45_4^{(1)}$	0.757259	-0.757158

The six globally optimum designs for Test Problem 2.

Design Rank	Stacking Sequence	Buckling Factor	Fitness
1	$90_4^\circ/\pm45_4^\circ/90_8^\circ$	3973.01	3973.01
2	$\pm45_2^\circ/90_{10}^\circ/\pm45_2^\circ/90_2^\circ$	3973.01	3973.01
3	$\pm45_2^\circ/90_8^\circ/\pm45_2^\circ/90_4^\circ$	3973.01	3973.01
4	$90_6^\circ/\pm45_2^\circ/90_2^\circ/\pm45_2^\circ/90_2^\circ/\pm45_2^\circ$	3973.01	3973.01
5	$90_2^\circ/\pm45_2^\circ/90_4^\circ/\pm45_2^\circ/0_2^\circ/90_2^\circ/\pm45_2^\circ$	3973.01	3973.01
6	$90_2^\circ/\pm45_2^\circ/90_8^\circ/\pm45_1^\circ/90_2^\circ$	3973.01	3973.01