ANL/MCS-TM-228

RECEIVED
SEP 08 1997
OSTI

Experiences with the PGAPack Parallel Genetic Algorithm Library

by

David Levine, Philip Hallstrom, David Noelle, Brian Walenz

MATHEMATICS AND COMPUTER SCIENCE DIVISION

Argonne National Laboratory, with facilities in the states of Illinois and Idaho, is owned by the United States government, and operated by The University of Chicago under the provisions of a contract with the Department of Energy.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Reproduced from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Prices available from (423) 576-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-228

Experiences with the PGAPack Parallel Genetic Algorithm Library

by

*David Levine,** *Philip Hallstrom,*† *David Noelle,*‡ *Brian Walenz*§

Mathematics and Computer Science Division

Technical Memorandum No. 228

July 1997

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

*Current address: Boeing Information & Support Services, P.O. Box 3707, MS 7L-20, Seattle, WA 98124-2207
†Current address: Dunn Systems, Inc., Lincolnwood, IL 60646
‡Current address: Computer Science Department, UCLA, Los Angeles CA 90024
§Current address: Sandia National Laboratories, Org. 9223, MS 1110, Albuquerque, NM 87185-1110

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Contents

Abstract 1

1 Introduction 1

2 Design and Implementation 2

3 Parallel Computing 4

4 Application Experiences 5

4.1 Molecular Docking 5

4.2 Quantum Chemistry Parameter Optimization 6

4.3 Timber Harvest Scheduling 7

4.4 Vehicle Clustering 7

4.5 Evolutionary Robotics 8

4.6 Finite Element Mesh Optimization 8

5 Conclusions and Future Work 9

Acknowledgments 10

References 10

Experiences with the PGAPack Parallel Genetic Algorithm Library

by

David Levine, Philip Hallstrom, David Noelle, Brian Walenz

Abstract

PGAPack is the first widely distributed parallel genetic algorithm library. Since its release, several thousand copies have been distributed worldwide to interested users. In this paper we discuss the key components of the PGAPack design philosophy and present a number of application examples that use PGAPack.

1 Introduction

PGAPack is the first widely distributed parallel genetic algorithm library. Since its release, several thousand copies have been distributed worldwide to interested users. Key features in PGAPack are support for multiple data types; parallel portability across uniprocessors, multiprocessors, and workstation networks; Fortran and C interfaces; a simple interface for novice and application users; multiple levels of access for expert users; object-oriented design; extensibility; and multiple GA operators and parameter choices.

PGAPack supports parallel and sequential implementations of the single population model. The population may be updated by using either generational or steady-state replacement schemes, or any of their parameterized variants. The supported crossover operators are one-point, two-point, and uniform crossover. Proportional, stochastic universal, binary tournament, or probabilistic binary tournament selection may be used. Different mutation operators are used with each different data type. A restart operator is available that reseeds a population by using random variants of the best string.

Options allow the user to specify the population size and stopping criteria and to whether duplicate strings should be allowed in the population and whether to mutate *or* crossover strings or to mutate *and* crossover strings. In all cases, defaults are provided if the user does not explicitly specify a choice. PGAPack system calls provide, random number generation, output control, error reporting, and debugging capabilities.

In simplest form, a parallel (or sequential) PGAPack program can be written by using only four PGAPack functions and a string evaluation function. Figure 1 shows such a minimal program and evaluation function for the Maxbit problem. `PGACreate` initializes PGAPack and returns the address of the context variable (see Section 2). The parameters to `PGACreate` are the program arguments, the data type (`PGA_DATATYPE_BINARY`), the string length (`100`), and the direction of optimization (`PGA_MAXIMIZE`). `PGASetUp` initializes all parameters and function pointers that have not been explicitly set (none in this example) to default values. `PGARun` executes the genetic algorithm. Its second argument is the name of a user-defined function (`evaluate`) that will be called whenever a string evaluation is required. `PGADestroy` releases all memory allocated by PGAPack.

```
#include "pgapack.h"
double evaluate (PGAContext *ctx, int p, int pop);
int main(int argc, char **argv)
{
    PGAContext *ctx;
    ctx = PGACreate (&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
    PGASetUp        (ctx                                                 );
    PGARun          (ctx, evaluate                                       );
    PGADestroy      (ctx                                                 );
    return;
}
double evaluate (PGAContext *ctx, int p, int pop)
{
    int i, nbits=0, stringlen;
    stringlen = PGAGetStringLength(ctx);
    for (i=0; i<stringlen; i++)
        if (PGAGetBinaryAllele(ctx, p, pop, i))
            nbits++;
    return((double) nbits);
}
```

Figure 1: PGAPack C Program for the Maxbit Example

The evaluation function (`evaluate`) must be written by the user. `PGAGetStringLength` returns the string length. `PGAGetBinaryAllele` returns the value of the `i`th bit of string `p` in population `pop`. The values returned by this function, sometimes called "raw fitness," are automatically mapped into nonnegative values according to whether any of the raw fitness values are negative, the direction of optimization, and the type of fitness function (identity, ranking, or linear normalization) used.

PGAPack provides functions to encode and decode real and integer values in a binary string. The string representation may be either binary or Gray coded. This capability allows the use of existing real- and integer-valued functions with no modification required to the function source. For example, suppose the user has a real-valued function f of three real variables x_1, x_2, and x_3, each constrained on the interval $[-10, 10]$, and wishes to use 10 bits for each and a Gray code encoding. This may be done as show in Figure 2. Note that the function f *need not be modified.* The function `grayfunc` is used as a "wrapper" to decode the real values from the Gray coded string, pass them as real values to f, and return the corresponding function value.

2 Design and Implementation

PGAPack supports four *native* data types: binary-valued, integer-valued, real-valued, and character-valued strings. A data-hiding capability provides the full functionality of the library to the user, in a transparent manner, irrespective of the data type used. The *context variable* is the data structure that provides the data hiding capability. The context variable is a pointer to a C language structure, which is itself a collection of other structures. These (sub)structures contain all

```
#include "pgapack.h"
double grayfunc (PGAContext *ctx, int p, int pop);
double f        (double x1, double x2, double x3);
 :   :
double grayfunc (PGAContext *ctx, int p, int pop)
{
    double x1, x2, x3, v;
    x1 =  PGAGetRealFromGrayCode (ctx, p, pop,  0,  9, -10., 10.);
    x2 =  PGAGetRealFromGrayCode (ctx, p, pop, 10, 19, -10., 10.);
    x3 =  PGAGetRealFromGrayCode (ctx, p, pop, 20, 29, -10., 10.);
    v  =  f(x1,x2,x3);
    return(v);
}
```

Figure 2: Using Legacy Code for Function Evaluation

the information necessary to run the genetic algorithm, including the data type to use, parameter values, the functions to call, operating system parameters, debugging flags, initialization choices, and internal scratch arrays.

All the functionality of PGAPack is provided through function calls. Typically, users call high-level, data-type-*neutral* functions, which themselves call data-type-*specific* functions that correspond to the data type used. The data-type-specific functions use addresses and offsets of the population data structures. The user-level routines, however, provide the abstraction of data type neutrality and an integer indexing scheme and can be called independent of the data type.

PGAPack maintains two populations: an *old* one and a *new* one. Formally, string p in population *pop* is referred to by the 2-tuple `(p,pop)` and the value of gene i in that string by the 3-tuple `(p,pop,i)`. To aid the user abstractions, two symbolic constants, `PGA_OLDPOP` and `PGA_NEWPOP`, always refer to the last generation and new generation, respectively.

PGAPack provides multiple levels of control to support the requirements of different users. The simplest level, shown in Figure 1, encapsulates the genetic algorithm "machinery" within the single function `PGARun`. The user need specify only three parameters: the data type, the string length, and the direction of optimization. All other parameters have default values.

At the next level, the user calls data-type-neutral functions explicitly (e.g., `PGASelect`, `PGACrossover`, `PGAMutation`). This mode is useful when the user wishes more explicit control over the steps of the genetic algorithm or wishes to hybridize the genetic algorithm with a hill-climbing heuristic.

At the third level, the user can customize the genetic algorithm by supplying function(s) to customize a particular operator(s) while still using one of the native data types. Finally, at the lowest level of usage, the user can define a *new* data type, write the data-type-specific low-level GA functions (e.g., crossover, mutation), and have these functions executed by the high-level data-type-neutral functions.

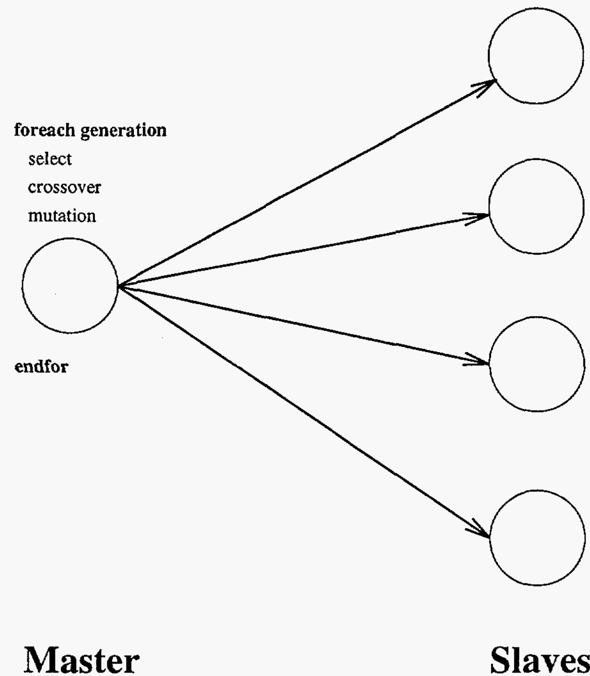

Figure 3: PGAPack Master-Slave Model

PGAPack is written in ANSI C. A set of interface functions allows user-level PGAPack functions to be called from Fortran. Message-passing calls follow the MPI (see Section 3) standard. Nonoperative versions of the MPI functions are supplied if the user does not have an MPI implementation for his machine. These allow the PGAPack library to be built (for sequential use) in the absence of an MPI implementation.

3 Parallel Computing

The first release of PGAPack was targeted primarily at application developers and supports a parallel (and sequential) implementation of the single population model. This initial choice was made because in most real applications, the dominant computational cost is executing function evaluations. Being able to execute these in parallel should significantly reduce the elapsed computing time.

The single population model may be parallelized by executing in parallel the loop iterations that create generation $t+1$ from generation t. Most steps in this loop—crossover, mutation, evaluation—can be executed in parallel. The execution efficiency, however, depends upon the computer architecture and parallel execution overhead, the number of new population members created each generation (the degree of parallelism), and the computational cost of the steps being executed in parallel (the granularity).

The parallel implementation in PGAPack uses a master/slave algorithm in which one process, the *master*, executes all steps of the genetic algorithm except the function evaluations, which are executed by *slave* processes. A master/slave implementation is shown in Figure 3.

We chose a master/slave algorithm for two reasons. First, since function evaluation time is the dominant cost in most GA runs, the performance benefits from parallel execution may be achieved by parallelizing only this step. Second, since we use a message-passing programming model to implement the master/slave algorithm, there may be a significant parallel execution overhead; focusing only on parallelizing the function evaluations allows for modest data distribution requirements (just the strings to be evaluated) and minimal synchronization requirements.

PGAPack is implemented by using the message-passing interface (MPI) standard [5]. MPI is a *specification* of a message-passing library for parallel computers and workstation networks; it defines a set of functions and their behavior. Implementations of MPI exist for both sequential (uniprocessors) and parallel (multiprocessors, multicomputers, and workstation networks) computer hardware, thereby allowing PGAPack to run on all these machines without any code changes. MPI offers a number of useful features that were used in PGAPack including collective communication operations, routines to configure the logical topology of the processors, barriers, a unique message namespace for library messages, and the ability to send and receive arbitrary structures.

The parallel implementation will produce the *same* result as the sequential implementation, usually faster. The choice of sequential or parallel execution depends on the number of processes specified when the program is started. If one process is specified, a sequential implementation is used. If two processes are used, both the master process and the slave process will compute the function evaluations. If more than two processes are used, the master executes all GA steps except the function evaluations, and the slaves execute the function evaluations.

There are two primary considerations in determining the performance advantage of using the master/slave model. First, the speedup will vary according to the amount of computation associated with a function evaluation and the computational overhead of distributing and collecting information to and from the slave processes. Second, the number of function evaluations that can be executed in parallel will limit the speedup. This number depends on the population size and the number of new strings created each generation. In a generational replacement model, the entire population may be evaluated in parallel. In the more popular steady-state model, however, typically only one or two new strings are produced, and the degree of parallelism is minimal. By default, PGAPack replaces 10% of the population. In our experience this percentage usually provides an acceptable degree of parallelism, while retaining the superior performance characteristics of the steady-state model.

4 Application Experiences

In this section we present examples of several projects that have used PGAPack. Our focus is on parallel execution and custom extensions to PGAPack.

4.1 Molecular Docking

STALK [7] is a system for molecular docking that uses PGAPack. The goal in molecular docking is to predict the conformation (location and orientation) of a ligand (a small molecule) in a protein active site (the part of the protein that the ligand binds to).

Table 1: Solution Time vs. No. of Processors

Compute Proc.	Time (sec.)	Speedup
1	263581	1.0
2	148666	1.8
3	87208	3.0
6	46950	5.6
13	22150	11.9
25	12831	20.5
50	7193	36.6
100	4181	63.0

Molecular docking may be formulated as a nonlinear optimization problem, where the goal is to maximize the intermolecular interaction energy. Typically, the solution space has a large number of possible conformations and many local minima. The energy function is highly nonlinear and computationally expensive to evaluate; being able to compute these in parallel is highly desirable for solving realistic problems.

STALK uses a rigid backbone model. The protein is fixed in space, and the position of the ligand determined by the GA. The six degrees of freedom of the ligand are translation and rotation about the protein's center of mass.

Table 1 contains performance results for a test problem with approximately 300 ligand atoms and 1500 protein atoms. The *Compute Proc.* column is the number of IBM SP processors that execute function evaluations. The *Time* column is the average over six runs of the total time spent by the master process (executing the GA, packing and sending data to the slave processes, and waiting for results). The *Speedup* column is the ratio of the time to execute the one-processor case to the time to execute with that number of processors. The speedup achieved is fairly constant, although not ideal. Several solutions with better energy values than the x-ray crystallographic solution were found.

A novel feature of STALK is the use of virtual-reality (VR) technology to provide an interactive computational steering capability. In the CAVE [4], a room-sized VR environment that communicates with the IBM SP through an SGI Onyx workstation front-end, the best conformation in the population and associated energy are displayed. Using a pointing device in the CAVE, the user may translate and/or rotate the ligand. The updated ligand coordinates are sent to the IBM SP running the GA which uses the GA's evaluation function to calculate and return the corresponding intermolecular energy. The user then has the option of using the "hand-docked" solution to replace the worst conformation in the population, to reseed the entire population using random perturbations of the hand-docked solution, or to make no changes at all.

4.2 Quantum Chemistry Parameter Optimization

In [3] PGAPack was used to study chemical reactions in the condensed phase. The goal was to develop a quantum mechanical (QM) simulation code that could determine accurate values for certain molecular properties (e.g., the heat of formation, dipole moments, atomic bond lengths, dihedral angles).

To calibrate the parameters of the QM simulation code, a GA was used to find values for these parameters that resulted in calculated molecular properties that were in close agreement with the experimentally determined results. Formally, if $Y_p(exp)$ is the value for property p known from the experimentally determined results and $Y_p(calc)$ the value for property p to be calculated, then the evaluation function is to minimize the sum of weighted errors given by

$$\sum_M \sum_p |Y_p(calc) - Y_p(exp)| \, w_p$$

where w_p is a weighting factor.

As a test case, the energetics of a proton transfer reaction in gas-phase and aqueous solution were studied. The basis set of molecules consisted of methanol, imidazole, methoxide, and imidazolium. A real-valued GA was run for 15,000 steady-state iterations. Parallelism was applied *within* each individual function evaluation by determining $Y_p(calc)$ for each molecule independently. Message passing was used to distribute and collect the function evaluation components. Using the parameter values determined by the GA, we were able to calculate more accurate values for the molecular properties of the proton transfer reaction with the QM simulation code than had been previously determined by other methods.

4.3 Timber Harvest Scheduling

The goal in adjacency constrained timber harvest scheduling (ACTHS) is to find near optimal tree harvesting schedules, subject to constraints on the clear-cut opening size and the level of timber harvests from schedule period to period. ACTHS is a difficult combinatorial optimization problem; a problem with N stands and M planning periods has M^N integer solutions (including infeasible solutions). A typical operational scheduling problem has 10–20 periods and 500–1000 stands.

In [8] the authors compare a traditional GA that represents solutions directly on the chromosomes with an order-based GA that represents permutations of the stand identification numbers on the chromosomes and uses each permutation as an order list for scheduling stands, with Monte Carlo integer programming (MCIP). PGAPack was used to implement both the traditional and order-based GAs. For the order-based GA, custom PGAPack operators were developed for order-based crossover, position-based crossover, partially matched crossover, order-based mutation, position-based mutation, and scramble mutation.

Test data ranged from 42 to 849 stands and 10 to 15 time periods. The results showed the order-based GA was superior to the other methods, averaging 2.2% better than the traditional GA, and 3.5% better than the MCIP. This project was so successful that the order-based GA is now the in-house production planning system for ACTHS at Rayonier Corp.

4.4 Vehicle Clustering

In [9] the authors describe the application of PGAPack to the Multiple-Depot Vehicle Routing Problem (MDVRP). The MDVRP is an extension of the vehicle routing problem, with the customers being served from multiple depots instead of a single, central depot. The MDVRP is solved by using a genetic clustering method that clusters customers using route primitives.

The genetic clustering is done using PGAPack. Once the clusters for the customers are obtained, the clusters are improved by using a branch-exchange procedure. The final solution obtained by the branch exchange procedure serves as the fitness value for the string. Because of the computationally expensive nature of the branch exchange procedure, a hash function was written for PGAPack that prevents previously evaluated strings from being evaluated again. The addition of the hash function reduced the processing time approximately 30%.

4.5 Evolutionary Robotics

The work in [2] describes the use of genetic algorithms to design neural network controllers for a simulated, box-pushing robot. The world of the robot is a small grid with blocks randomly placed on it. The goal is to have the robot push blocks into the corners, placing blocks near the corner blocks when the corners are filled.

The evaluation function for this problem is very expensive. Each robot must be evaluated in more than one initial configuration, and each configuration requires several hundred time steps. A parallel version of PGAPack was installed on an SGI workstation network and used to corroborate previous results.

In addition, a version of niching based on the work in [6] was added to PGAPack to support multi-objective optimization. A custom function was written that performed selection by choosing two strings randomly from the population and returning the one which dominates a random sample of the population. If neither string dominates, the string with the fewest neighbors is returned.

4.6 Finite Element Mesh Optimization

In finite-element analysis, structures are modeled as meshes of elements and nodes that match the geometry, rigidity, and loading of each structure. While a very fine uniform mesh will give accurate results, it usually leads to unacceptably high computational loads. Therefore, a nonuniform mesh with higher resolution is used in parts of the structure where the stress gradients are high, and with lower resolution where the stress gradients are low is desired.

In the work in [1], a GA is used to search for an optimal mesh. Initially, a uniform mesh is imposed on a loaded structure with a small number of degrees of freedom that are not computationally burdensome. An energy-based error norm is calculated and used as an objective function to be minimized. In the main loop, randomly perturbed node positions are generated, the finite-element mesh is regenerated by using the new node positions, and the objective function is recalculated. The mesh regeneration is stopped when the objective function has reached a stationary (assumed to be minimum) value.

This approach has been implemented on simple beams and plates by using PGAPack on a cluster of eight Sun workstations. The results appear promising for the simple cases tried. A real-coded GA is used that has a specialized function to regenerate elements around the perturbed positions of the nodes. This function eliminates all meshes that result in malformed (e.g., high aspect ratio, concave or physically impossible) elements before mutation and crossover are attempted.

5 Conclusions and Future Work

The number of users and the applications they have implemented underscores the fact that PGAPack has met with widespread acceptance. We attribute this success to four key factors. First is ease of use. Many users use PGAPack as a black-box; parameter and operator choices have robust default values, and their details are encapsulated in a few simple function calls. The object-oriented interface, coupled with the user-level abstractions, allows the user to concentrate on functionality and not data structure details.

The second important attribute is portability. PGAPack has been successfully installed on most workstations, workstation clusters, and parallel computers. The ANSI C language PGAPack is written in is standardized and fully portable. A small set of link options provides a compatible Fortran interface. From a parallel computing perspective, the message-passing programming model maps easily onto both shared- and distributed-memory hardware. Additionally, MPI is now a fully accepted message-passing standard, with versions available for all current workstations and parallel computers.

The third reason for PGAPack's success is the task parallel master/slave model. All real applications we have worked on, and most others we are aware of, are dominated by the computational cost of the function evaluations. The performance improvement from executing these evaluations simultaneously can provide significantly improved turnaround. In some cases the improved performance is critical to being able even to apply GAs.

Finally, the fourth reason for PGAPack's success is extensibility. Although PGAPack supports multiple data types and their common operators, problem-specific functionality is sometimes needed. PGAPack provides a simple means to replace any operator with a user function. Also, users may define a new data type by writing the low-level data-structure-specific functions, but still take advantage of the high-level data-structure-neutral functions in PGAPack. Finally, by passing the context variable as a parameter to a user function, the user has complete access to solution and parameter values, and may develop any custom functionality desired.

Prototype implementations of several new features have been developed for future incorporation into PGAPack. These include an island model GA, a genetic programming implementation, and a meta-GA for optimizing parameter choices. In addition, we plan to incorporate the additional functionality that has been developed and contributed by our users.

PGAPack is freely available and may be obtained by anonymous ftp from `info.mcs.anl.gov` in file `pub/pgapack/pgapack.tar.Z`, or via the World Wide Web at the following URL: `http://www.mcs.anl.gov/home/levine/PGAPACK/index.html`.

Acknowledgments

We thank Ara Arabyan, Karthik Balakrishnan, Ralph Butler, David Mullen, and Sam Thangiah for discussions of their PGAPack applications; Greg Reeder for writing prototype MPI communication routines; and Bill Gropp, Lois Curfman McInnes, and Barry Smith for many helpful discussions about their PETSc library.

References

[1] A. Arabyan, S. Chemishkian, and A. Tonoyan. Finite-element mesh optimization using genetic algortihms. Univeristy of Arizona, 1997.

[2] K. Balakrishnan and V. Honavar. Analysis of neurocontrollers designed by simulated evolution. In *Proceedings of IEEE International Conference on Neural Networks ICNN'96*, 1996.

[3] P. Bash, L. Ho, A. MacKerell, P. Hallstrom, and D. Levine. Progress toward chemical accuracy in the computer simulation of condensed phase reactions. *Proceedings of the National Academy of Sciences*, 93:3698–3703, 1996.

[4] C. Cruz-Neira, D. Sandin, and T. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the CAVE. In *ACM SIGGRAPH '93 Proceedings*, pages 135–142. Lawrence Erlbaum Associates, 1993.

[5] W. Gropp, E. Lusk, and A. Skjellum. *USING MPI Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, 1994.

[6] J. Horn and N. Nafpliotis. Multiobjective optimization using the niched pareto genetic algorithm. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, volume 1, Piscataway, 1994. Lawrence Erlbaum Associates.

[7] D. Levine, M. Facello, P. Hallstrom, G. Reeder, B. Walenz, and F. Stevens. STALK: An Interactive Virtual Molecular Docking System, 1996. Accepted for publication in *IEEE Computational Science & Engineering.*

[8] D. Mullen. A comparison of genetic algorithms and monte carlo integer programming for optimization of adjacency constrained timber harvest scheduling problems. Master's thesis, University of North Florida, 1996.

[9] S. Salhi, S. Thangiah, and F. Rahman. A genetic clustering method for the multi-depot vehicle routing problem. In *Proceedings of the Third International Conference on Neural Networks and Genetic Algorithms*, 1997.