

Distribution Category:
Mathematics and
Computer Science (UC-405)

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL-95/18

Users Guide to the PGAPack Parallel Genetic Algorithm Library

by

David Levine

Mathematics and Computer Science Division

and contributors

January 31, 1996, last updated May 31, 2020

This work was supported 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.

This is not the original report, it contains small fixes reported by users and documentation updates for new features.

Acknowledgments

Much of the code in PGAPack was originally developed as part of the author's Ph.D. thesis. Significant contributions to the development of PGAPack were made by Philip Hallstrom, David Noelle, Greg Reeder, and Brian Walenz, participants in Argonne's Science and Engineering Research Semester program.

Many aspects of PGAPack—including the user interface, choice of some data structures, and design of Fortran wrappers—were strongly influenced by the design of the PETSc (Portable and Extensible Tools for Scientific Computing) library. I thank Bill Gropp, Lois Curfman McInnes, and Barry Smith for many helpful discussions. The code in PGAPack for parsing command line arguments is a modified version of that used in the `p4` system developed by Ralph Butler and Rusty Lusk.

Contents

0	Quick Start	1
I	Getting Started	2
1	Introduction	3
2	Examples	4
2.1	Maxbit Problem in C	4
2.2	Maxbit Problem in Fortran	4
2.3	Specifying Nondefault Values	6
2.4	Differential Evolution	6
2.5	Parallel I/O	7
2.6	Compiling, Linking, and Execution	7
II	Users Guide	11
3	The Structure of PGAPack	12
3.1	Native Data Types	12
3.2	Context Variable	12
3.3	Levels of Usage Available	12
3.4	Function Call-Based Library	13
3.5	Header File and Symbolic Constants	13
3.6	Evaluation Function	13
3.7	Parallelism	13
3.8	Implementation	14
4	Basic Usage	15
4.1	Required Functions	15
4.2	Population Replacement	16
4.3	Stopping Criteria	20
4.4	Initialization	20
4.5	Selection	21
4.6	Crossover	22
4.7	Mutation	22
4.8	Restart	25
4.9	String Evaluation and Fitness	25
4.10	Accessing Allele Values	27
4.10.1	Representing an Integer with a Binary String	28
4.10.2	Representing a Real Value with a Binary String	29
4.10.3	Example	29
4.11	Report Options	30

4.12	Utility Functions	30
4.12.1	Random Numbers	30
4.12.2	Print Functions	31
4.12.3	Miscellaneous	31
4.13	Command-Line Arguments	31
5	Explicit Usage	32
5.1	Notation	32
5.2	Simple Sequential Example	32
5.3	Complex Example	33
5.4	Explicit PGAPack Functions	35
6	Custom Usage: Native Data Types	36
6.1	Basics	36
6.2	Example Problem: C	38
6.3	Example Problem: Fortran	38
7	Custom Usage: New Data Types	41
7.1	Basics	41
7.2	Example Problem	41
8	Hill-Climbing and Hybridization	48
9	Parallel Aspects	50
9.1	Basic Usage	50
9.2	Explicit Use	50
9.3	Example	51
9.4	Performance	51
10	Fortran Interface	53
11	Debugging Tools	55
III	Appendixes	58
A	Default Values	59
B	Function Bindings	61
C	Parallelism Background	68
D	Machine Idiosyncrasies	72
E	Common Problems	75
	Bibliography	77

Chapter 0

Quick Start

If you wish to get started by just typing a few lines and running an example, this section is for you. We trust you know how to check out the latest version from github. Once you have a directory with the checked-out version you can build with:

1. `cd pgapack`
2. `make`

The Makefile will auto-detect if you have an MPI-Implementation installed and will build a parallel version. If no MPI is detected, the serial version will be built. If the version is not correctly auto-detected or you want to force a certain MPI backend, refer to the build documentation in `README.rst`.

Chapter 2 (example problems) and Sections 4.1 (required functions) and 4.9 (string evaluation and fitness) should be read next.

Part I

Getting Started

Chapter 1

Introduction

PGAPack is a parallel genetic algorithm library that is intended to provide most capabilities desired in a genetic algorithm package, in an integrated, seamless, and portable manner. Key features of PGAPack are as follows:

- Ability to be called from Fortran or C.
- Executable on uniprocessors, multiprocessors, multicomputers, and workstation networks.
- Binary-, integer-, real-, and character-valued native data types.
- Object-oriented data structure neutral design.
- Parameterized population replacement.
- Multiple choices for selection, crossover, and mutation operators.
- Easy integration of hill-climbing heuristics.
- Easy-to-use interface for novice and application users.
- Multiple levels of access for expert users.
- Full extensibility to support custom operators and new data types.
- Extensive debugging facilities.
- Large set of example problems.

Chapter 2

Examples

This chapter presents some simple PGAPack programs. The problem chosen is the Maxbit problem. The objective is to maximize the number of 1-bits in a string.

Section 2.1 presents a simple PGAPack program in C whose structure is sufficient to solve many problems. Section 2.2 presents this same program in Fortran. Section 2.3 shows how to change default values in PGAPack. Section 2.5 contains an example that shows how keyboard input may be read in an MPI environment. Finally, Section 2.6 shows how to compile, link, and execute a PGAPack program. These and other examples may be found in the `./examples/c` and `./examples/fortran` directories.

2.1 Maxbit Problem in C

Figure 2.1 shows a minimal program and evaluation function in C for the Maxbit problem. All PGAPack C programs *must* include the header file `pgapack.h`. The `PGACreate` call is always the first function called in a PGAPack program. It initializes the context variable, `ctx`. The parameters to `PGACreate` are the arguments to the program (given by `argc` and `argv`), the data type selected (`PGA_DATATYPE_BINARY`), the string length (100), and the direction of optimization (`PGA_MAXIMIZE`). The `PGASetUp` call initializes all parameters and function pointers not explicitly set by the user to default values.

`PGARun` executes the genetic algorithm. Its second argument is the name of a user-defined function (`evaluate`) that will be called to evaluate the strings. `PGADestroy` releases all memory allocated by PGAPack. Note that all PGAPack functions take the context variable as an argument (except `PGACreate`, which creates the context variable).

The `evaluate` function must be written by the user, must return a `double`, and must follow the exact calling sequence shown. An evaluation function may return more values than just the return value in the array pointed to by `aux`. This can be used for evaluating constraints for constrained problems or for multi-objective optimization. Usually the number of auxiliary return values is zero and the `aux` argument is ignored. For details, see section 4.9. `PGAGetStringLength` returns the string length. `PGAGetBinaryAllele` returns the value of the `i`th bit of string `p` in population `pop`.

2.2 Maxbit Problem in Fortran

The Fortran Maxbit problem in Figure 2.2 is similar to the C version in Figure 2.1. The Fortran include file is `pgapackf.h` and should be included in every Fortran function or subroutine that makes PGAPack calls¹. Since Fortran provides no standard mechanism for specifying command line arguments, these are omitted from the `PGACreate` function call. The context variable, `ctx`, is declared `integer` in Fortran.

The evaluation function `evaluate` must contain exactly the calling sequence shown and must return a `double precision` value. The `aux` value is optional in Fortran because Fortran does less strict type-checking than C and with standard calling conventions the argument can be omitted if not needed. Note that `evaluate`

¹Since not all Fortran compilers support the `-I` mechanism for specifying the include file search path, you will need to copy or set up a symbolic link to `pgapackf.h` from the directory you are compiling a Fortran program in.

```

#include "pgapack.h"
double evaluate (PGAContext *ctx, int p, int pop, double *aux);

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, double *aux)
{
    int i, nbits, stringlen;

    stringlen = PGAGetStringLength(ctx);
    nbits     = 0;
    for (i=0; i<stringlen; i++)
        if (PGAGetBinaryAllele(ctx, p, pop, i))
            nbits++;
    return((double) nbits);
}

```

Figure 2.1: PGAPack C Program for the Maxbit Example

```

include "pgapackf.h"
external evaluate
integer ctx
ctx = PGACreate (PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE)
call PGASetUp (ctx                                     )
call PGARun   (ctx, evaluate                           )
call PGADestroy(ctx                                     )
stop
end

double precision function evaluate (ctx, p, pop)
include "pgapackf.h"
integer ctx, p, pop, i, bit, nbits, stringlen
stringlen = PGAGetStringLength(ctx)
nbits     = 0
do i=1, stringlen
    bit = PGAGetBinaryAllele(ctx, p, pop, i)
    if (bit .eq. 1) then
        nbits = nbits + 1
    endif
enddo
evaluate = dble(nbites)
return
end

```

Figure 2.2: PGAPack Fortran Program for the Maxbit Example

```

#include "pgapack.h"
double evaluate (PGAContext *ctx, int p, int pop, double *aux);

int main(int argc, char **argv)
{
    PGAContext *ctx;
    ctx = PGACreate      (&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
    PGASetPopSize        (ctx, 500                               );
    PGASetFitnessType     (ctx, PGA_FITNESS_RANKING              );
    PGASetCrossoverType   (ctx, PGA_CROSSOVER_UNIFORM           );
    PGASetUp              (ctx                                   );
    PGARun                (ctx, evaluate                         );
    PGADestroy            (ctx                                   );
    return;
}

```

Figure 2.3: Specifying Nondefault Values

is declared in an **external** statement in the program unit in which it is used as an actual argument. This is a requirement of the Fortran language. In Fortran, the range of allele values is `1:stringlen`, rather than `0:stringlen-1` as in C.

2.3 Specifying Nondefault Values

PGAPack offers a wide range of choices for parameter values, operators, and algorithmic choices. These will be set to default values in `PGASetUp` if the user does not explicitly set a value for them. A nondefault value may be set by using the `PGASet` family of calls *after* `PGACreate` has been called, but *before* `PGASetUp` has been called.

In Figure 2.3 the `PGASet` calls change the default values for population size, fitness calculation, and crossover type. `PGASetPopSize` changes the population size to 500. `PGASetFitnessType` specifies that the fitness values be determined by using a ranking procedure rather than by direct use of the evaluation function values. `PGASetCrossoverType` specifies that uniform crossover, rather than the default of two-point crossover is to be used. Most `PGASet` calls are discussed in Chapter 4.

2.4 Differential Evolution

Differential Evolution (DE) is an evolutionary algorithm (EA) invented by Price and Storn in the 1990's [26, 27, 22]. It is used with floating-point genes and uses differences of individuals (floating-point vectors) which are added to another vector to form a *donor* vector which is then crossed-over with an existing individual. The algorithm is described in more detail in section 4.7, page 23. Since in PGAPack the DE algorithm is implemented in a mutation strategy, typically for DE a strategy with only mutation is selected, see `PGASetMixingType` with option `PGA_MIX_MUTATE_ONLY` in section 4.2.

DE applies selection pressure during population replacement: A newly-mutated string replaces its parent if it has the same or a better fitness. There is no selection mechanism during the selection phase like in other EAs. To emulate this (non-) selection, PGAPack introduces a new selection type, linear selection, which just returns all individuals in sequence and is no selection operator in the genetic-algorithm sense because no selection pressure is applied. More details of the selection operator for DE are given in section 4.5.

For the population replacement strategy the pairwise-best replacement type is introduced for DE, which can also be used in other EA variants due to the modular nature of PGAPack, more details are given in section 4.2. Typical settings for Differential Evolution are given in figure 2.4. In that example the population size and the number of individuals replaced in each generation are set to the same value: Since

```

PGASetPopSize (ctx, 30);
PGASetNumReplaceValue (ctx, 30);

PGASetSelectType (ctx, PGA_SELECT_LINEAR);
PGASetPopReplaceType (ctx, PGA_POPREPL_PAIRWISE_BEST);
PGASetMixingType (ctx, PGA_MIX_MUTATE_ONLY);
PGASetMutationType (ctx, PGA_MUTATION_DE);
PGASetMutationBounceBackFlag (ctx, PGA_TRUE);

```

Figure 2.4: Specifying Nondefault Values for Differential Evolution

DE's replacement strategy replaces individuals only if they are better than an existing individual this strategy is elitist and it makes sense to apply DE to all individuals in each generation.

2.5 Parallel I/O

The examples in Figures 2.5 (C) and 2.6 (Fortran) read values for the two parameters `len` (string length) and `maxiter` (maximum number of GA iterations) from standard input. These examples will work correctly with either a sequential or parallel version of PGAPack. However, the explicit use of MPI calls for I/O is necessary *only* if a parallel version of PGAPack is used, and parameter values are read from standard input. The purpose is to be sure that each process receives a copy of the input values. See Appendix C for further details.

`MPI_Init(&argc, &argv)` is always the first function called in any MPI program. Each process executes `MPI_Comm_rank(MPI_COMM_WORLD, &myid)` to determine its unique rank in the communicator² `MPI_COMM_WORLD`. The logic used in this program is to have process 0 read and write from/to standard input/output and broadcast (using `MPI_Bcast`) the parameters to the other processes. The PGAPack function calls are similar to those in the previous examples. If the user called `MPI_Init`, the user must also call `MPI_Finalize` before exiting.

We elaborate here on the `MPI_Bcast` function because of its practical value in the model of parallel I/O shown. For more detailed discussion of MPI concepts and functions, the user should consult [13, 15].

The C binding for `MPI_Bcast` is

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

and the Fortran binding

```

MPI_BCAST(buffer, count, datatype, root, comm, ierror)
<type> buffer(*)
integer count, datatype, root, comm, ierror

```

`MPI_Bcast` will result in every process in communicator `comm` receiving a copy of the contents of `*buf`/`buffer`. The other parameters are the number of items (`count`), the datatype (`datatype`), which may be one of `MPI_DOUBLE`, `MPI_INT`, `MPI_CHAR`, `MPI_UNSIGNED`, or `MPI_LONG`; the rank of the process with the original copy (`root`); the MPI communicator (`comm`); and, for Fortran, a variable to handle an error return code (`ierror`).

2.6 Compiling, Linking, and Execution

When PGAPack was installed, the makefiles in the `./examples/c` and `./examples/fortran` directories were correctly configured for the machine PGAPack was installed on using the version of MPI specified (if any). To run your own programs, it is best to copy the appropriate makefile (C or Fortran) to your directory and modify it to use your source code files. The makefile will compile your source code files, link in the PGAPack library (and MPI library if a parallel version of PGAPack was built), and build your executable.

²See Appendix C

```

#include "pgapack.h"
double evaluate (PGAContext *ctx, int p, int pop, double *aux);

int main( int argc, char **argv )
{
    PGAContext *ctx;
    int myid, len, maxiter;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0) {
        /* Process 0 has a dialog */
        printf("String length? ");      /* with the user and */
        scanf("%d", &len);              /* broadcasts the user's */
        printf("Max iterations? ");      /* parameters to all */
        scanf("%d", &maxiter);          /* other processes */
    }
    MPI_Bcast(&len, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&maxiter, 1, MPI_INT, 0, MPI_COMM_WORLD);

    ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, len, PGA_MAXIMIZE);
    PGASetMaxGAIterValue(ctx, maxiter);
    PGASetUp(ctx);
    PGARun(ctx, evaluate);
    PGADestroy(ctx);

    MPI_Finalize();
    return(0);
}

```

Figure 2.5: PGAPack Maxbit Example in C with I/O

```

include 'pgapackf.h'
include 'mpif.h'

double precision evaluate
external          evaluate

integer ctx, myid, len, maxiter, ierror

call MPI_Init(ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierror)

c   Process 0 has a dialog with the user and broadcasts the user's
c   parameters to all other processes
if (myid .eq. 0) then
    print *, 'String length?'
    read *, len
    print *, 'Max iterations?'
    read *, maxiter
endif
call MPI_Bcast(len,      1, MPI_INT, 0, MPI_COMM_WORLD, ierror)
call MPI_Bcast(maxiter, 1, MPI_INT, 0, MPI_COMM_WORLD, ierror)

ctx = PGACreate(PGA_DATATYPE_BINARY, len, PGA_MAXIMIZE)
call PGASetMaxGAIterValue(ctx, maxiter)
call PGASetUp(ctx)
call PGARun(ctx, evaluate)
call PGADestroy(ctx)

call MPI_Finalize(ierror)

stop
end

```

Figure 2.6: PGAPack Maxbit Example in Fortran with I/O

How you execute your program will depend on whether a sequential or parallel version of PGAPack was built, the MPI implementation used and the machine you are running on. If a sequential version of PGAPack was built (i.e., one where the user did not supply a version of MPI), the executable `maxbit` can be executed on a uniprocessor Unix system by typing `maxbit`. If the `MPICH` implementation of MPI was used, it may be executed (using four processes) by `mpirun maxbit -np 4`. Appendix D contains some examples.

Part II

Users Guide

Chapter 3

The Structure of PGAPack

This chapter provides a general overview of the structure of PGAPack.

3.1 Native Data Types

PGAPack is a data-structure-neutral library. By this we mean that a data-hiding capability provides the full functionality of the library to the user, in a transparent manner, irrespective of the data type used. PGAPack supports four native data types: binary-valued, integer-valued, real-valued, and character-valued strings. In addition, PGAPack is designed to be easily extended to support other data types (see Chapter 6).

The binary (or bit) data type (i.e., `|1|0|1|1|`) is the traditional GA coding. The bits may either be interpreted literally or decoded into integer or real values by using either binary coded decimal or binary-reflected Gray codes. In PGAPack the binary data type is implemented by using each distinct bit in a computer word as a gene, making the software very memory-efficient. The integer-valued data type (i.e., `|3|9|2|4|`) is often used in routing and scheduling problems. The real-valued data type (i.e., `|4.2|7.1|-6.3|0.8|`) is useful in numerical optimization applications. The character-valued data type (i.e., `|h|e|l|l|o|w|o|r|l|d|`) is useful for symbolic applications.

3.2 Context Variable

In PGAPack 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 the information necessary to run the genetic algorithm, including data type specified, parameter values, which functions to call, operating system parameters, debugging flags, initialization choices, and internal scratch arrays. By hiding the actual data type selected and specific functions that operate on that data type in the context variable, user-level functions in PGAPack can be called independent of the data type.

Almost all fields in the context variable have default values. However, the user can set values in the context variable by using the `PGASet` family of function calls. The values of fields in the context variable may be read with the `PGAGet` family of function calls.

3.3 Levels of Usage Available

PGAPack provides multiple levels of control to support the requirements of different users. At the simplest level, the genetic algorithm “machinery” is encapsulated within the `PGARun` function, and 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 the data-structure-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 his or her own function(s) to provide a particular operator(s) while still using one of the native data types. Finally, the user can define his or her own datatype, write the data-structure-specific low-level GA functions for the datatype (i.e., crossover, mutation, etc.), and have the data-structure-specific functions executed by the high-level data-structure-neutral PGAPack functions.

3.4 Function Call-Based Library

All the access to, and functionality of, the PGAPack library is provided through function calls.

- The **PGASet** family of functions sets parameter values, allele values, and specifies which GA operators to use. For example, `PGASetPopSize(ctx,500)` sets the GA population size to 500.
- The **PGAGet** family of functions returns the values of fields in the context variable and allele values in the string. For example, `bit = PGAGetBinaryAllele(ctx,p,pop,i)` returns the value of the *i*th bit in string *p* in population *pop* into *bit*.
- The simplest level of usage is provided by the **PGARun** function. This function will run the genetic algorithm by using any nondefault values specified by the user and default values for everything else.
- The next level of usage is provided by the data-structure-neutral functions, which the user can call to have more control over the specific steps of the genetic algorithm. Some of these functions are **PGASelect**, **PGACrossover**, **PGAMutate**, **PGAEvaluate**, and **PGAFitness**.
- The data-structure-specific functions deal directly with native data types. In general, the user never calls these functions directly.
- System calls in PGAPack provide miscellaneous functionality, including debugging, random number generation, output control, and error reporting.

3.5 Header File and Symbolic Constants

The PGAPack header file contains symbolic constants and type definitions for all functions and should be included in any file (or function or subroutine in Fortran) that calls a PGAPack function. For example, `PGA_CROSSOVER_UNIFORM` is a symbolic constant that is used as an argument to the function `PGASetCrossoverType` to specify uniform crossover. In C the header file is `pgapack.h`. In Fortran it is `pgapackf.h`.

3.6 Evaluation Function

PGAPack requires that the user supply a function that returns an evaluation of a string that it will map to a fitness value. This function is called whenever a string evaluation is required. The calling sequence and return value of the function must follow the format discussed in Section 4.9.

3.7 Parallelism

PGAPack can be run on both sequential computers (uniprocessors) and parallel computers (multiprocessors, multicomputers, and workstation networks). The parallel programming model used is message passing, in particular the single program, single data (SPMD) model. PGAPack version 1.0 supports sequential and parallel implementations of the global population model (see Chapter 9).

3.8 Implementation

PGAPack is written in ANSI C. A set of interface functions allows most user-level PGAPack functions to be called from Fortran. All message-passing calls follow the Message Passing Interface (MPI) standard [13, 15]. Nonoperative versions of the basic MPI functions used in the examples are supplied if the user does not provide an MPI implementation for their machine. These routines simply return and provide *no* parallel functionality. Their purpose is to allow the PGAPack library to be built in the absence of an MPI implementation.

Most low-level internal functions in PGAPack are data-structure *specific* and use addresses and/or offsets of the population data structures. The user-level routines, however, provide the abstractions of data-structure *neutrality* and an integer indexing scheme for access to population data structures.

Chapter 4

Basic Usage

As the examples in Chapter 2 show, a PGAPack program can be written with just four function calls and a string evaluation function. This basic usage is discussed further in Section 4.1. Sections 4.3–4.12 explain options available in PGAPack. Section 4.13 discusses PGAPack command line arguments.

4.1 Required Functions

Any file (or function or subroutine in Fortran) that uses a PGAPack function must include the PGAPack header file. In C this file is `pgapack.h`. In Fortran this file is `pgapackf.h`. The first PGAPack call made is *always* to `PGACreate`. In C this call looks like

```
PGAContext *ctx;
ctx = PGACreate (&argc, argv, datatype, len, maxormin);
```

`PGACreate` allocates space for the context variable, `ctx` (Section 3.2), and returns its address. `argc` and `argv` are the standard list of arguments to a C program. `datatype` must be one of `PGA_DATATYPE_BINARY`, `PGA_DATATYPE_INTEGER`, `PGA_DATATYPE_REAL`, or `PGA_DATATYPE_CHARACTER` to specify strings consisting of binary-valued, integer-valued, real-valued, or character-valued strings, respectively. `len` is the length of the string (i.e., the number of genes). `maxormin` must be `PGA_MAXIMIZE` or `PGA_MINIMIZE` to indicate whether the user's problem is maximization or minimization, respectively.

In Fortran the call to `PGACreate` is

```
integer ctx
ctx = PGACreate (datatype, len, maxormin)
```

where `datatype`, `len`, and `maxormin` are the same as for C programs. After the `PGACreate` call, the user may *optionally* set nondefault values. These are then followed by a call to `PGASetUp` to initialize to default values all options, parameters, and operators not explicitly specified by the user. For example,

```
ctx = PGACreate(&argc, argv, datatype, len, maxormin);
PGASetPopSize      (ctx, 500);
PGASetFitnessType  (ctx, PGA_FITNESS_RANKING);
PGASetCrossoverType (ctx, PGA_CROSSOVER_UNIFORM);
PGASetUniformCrossoverProb (ctx, 0.6);
PGASetUp           (ctx);
```

will change the default values for the population size, the mapping of the user's evaluation to a fitness value, and the crossover type. All `PGASet` calls should be made *after* `PGACreate` has been called, but *before* `PGASetUp` has been called; all such calls are *optional*. Note also that *all* PGAPack functions other than `PGACreate` take the context variable as their first argument.

The `PGARun` function executes the genetic algorithm. Its second argument is the name of a user-supplied evaluation function that returns a `double` (`double precision` in Fortran) value that is the user's evaluation of an individual string. In C the prototype for this function looks like

```
double evaluate (PGAContext *ctx, int p, int pop, double *aux);
```

and in Fortran

```
double precision function evaluate (ctx, p, pop, aux)
integer ctx, p, pop
double precision aux(*)
```

The user *must* write the evaluation function, and it *must* have the calling sequence shown above and discussed further in Section 4.9, except that depending on the architecture and the calling convention of the compiler, the `aux` argument can be left out. After `PGARun` terminates, `PGADestroy` is called to release all memory allocated by `PGAPack`.¹

Except for writing an evaluation function (Section 4.9) the information contained in rest of this chapter is optional—defaults will be set for all other GA parameters. We do note, however, that the defaults used are the result of informal testing and results reported in the GA literature. *They are by no means optimal*, and additional experimentation with other values may well yield better performance on any given problem.

4.2 Population Replacement

PGAPack supports several different population replacement schemes. Among them the two most common replacement schemes in the literature. The first, the *generational replacement* genetic algorithm (GRGA), replaces the entire population each generation and is the traditional genetic algorithm [19]. The second, the *steady-state* genetic algorithm (SSGA), typically replaces only a few strings each generation and is a more recent development [28, 32, 33]. A third scheme, originally called *restricted tournament selection* by Harik [16, 17] and later adopted under the name of *restricted tournament replacement* by Pelikan [21] replaces offspring candidates into the original population by selecting a number of members from the original population and selecting the member most similar to the candidate. The similarity metric is implemented by a genetic difference function, see section 6.1, p. 37. The candidate is then compared to the most similar member and only if the new solution candidate is better than the member it replaces it. This approach is repeated for each new solution. A fourth approach used by evolutionary algorithm variants that mutate an individual into an offspring that replaces its parent only when it is better is also supported. This variant is used by the popular Differential Evolution [26, 27, 22] algorithm. We call this replacement variant *pairwise best* in the following. Individuals with the same index in the old and the new population are compared and the one with the better fitness is used.

Two algorithms are typically used for multi-objective optimization. The first one is the elitist Nondominated Sorting Genetic Algorithm (Version 2), NSGA-II [10] and is called NSGA-II replacement. It can be used for single-objective optimization, too, both with and without constraints. If constraints are present, by default the constraint violations are summed. An alternative is to use nondominated sorting for constraints, too. This can be switched on by setting `PGASumConstraintsFlag` to `PGA_FALSE`.

The second is the Nondominated Sorting Genetic Algorithm for many-objective optimization, NSGA-III [9], [20]. NSGA-II and NSGA-III are the only possible population replacement algorithms when using multi-objective optimization.

With NSGA-III you need to define a regular set of points or a set of directions where you want solutions to the multi-objective problem to be found, both can be combined, you can specify both, a number of reference directions and reference points. The reference points are in a hyperplane defined by the M positive axes of the objective space, the hyperplane goes through the axes intercepts at coordinate 1 for each of the axes. An example for three objectives with a partition size of 12 is shown in figure 4.1.

The discovered pareto-front is projected onto this hyperplane [2]. When specifying reference directions, these are directly defined in the objective space (without any projection).

To compute a set of points we use an algorithm originally defined in a paper by I. Das and J. E. Dennis [3] with the function `LIN_dasdennis`. This function gets the dimension (the number of objectives to optimize which is the number of auxiliary evaluations +1 minus the number of constraints) and the number of partitions. It returns the points in `result` and optionally takes a scale factor in the range 0..1 and a

¹`PGADestroy` will also call `MPI_finalize`, if `MPI` was started by `PGACreate`.

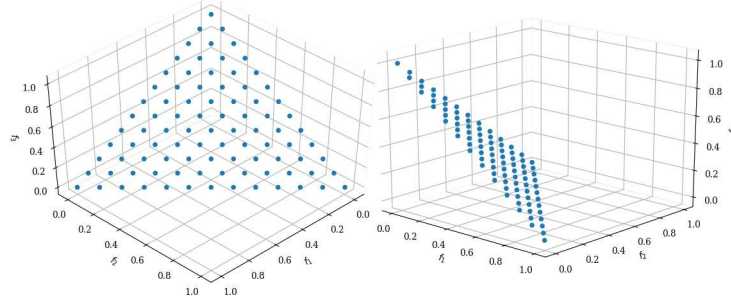


Figure 4.1: Reference points in 3 dimensions, partition-size 12

direction to shift this scaled set of points. The direction is only needed if the scale factor is less than one. The first time the function is called, the result must point at a NULL pointer. The function automatically allocates the necessary memory. It can be called multiple times to extend the points already allocated. The resulting points are then passed into the function `PGASetReferencePoints`:

```
int npoints = 0;
void *result = NULL;
double point [3] = {1, 1, 1};

PGASetNumAuxEval (ctx, 2);
PGASetNumConstraint (ctx, 0);
npoints = LIN_dasdennis (3, 2, &result, 0, 1, NULL);
npoints = LIN_dasdennis (3, 1, &result, npoints, 0.5, point);
PGASetReferencePoints (ctx, npoints, result);
```

For defining reference directions, the function `PGASetReferenceDirections` is used. It gets the number of directions and the vector of directions (each direction is a vector of the dimensionality of the number of objectives) and the number of partitions (for Das/Dennis points) and the scale factor of the generated points:

```
double directions [] [3] = {{1, 1, 1}, {1, 2, 3}};
PGASetReferenceDirections (ctx, 2, directions, 12, 0.05);
```

The difference to the reference points above is that the reference directions are in the objective space and the Das/Dennis points are generated dynamically in each generation.

Note that by default when no population size is specified, NSGA-III uses the number of points defined by the reference points and reference directions for the population size.

The NSGA-III replacement optimizes the solutions to be near the reference points and/or reference directions. With a high number of objective functions, the N-dimensional space forming the solution space increases exponentially with the number of objective functions. This is known as the “curse of dimensionality”. With NSGA-II it is increasingly hard to find a well distributed set of solutions with more than two or three objectives. With the NSGA-III replacement it is possible to concentrate the search to a predefined number of reference points or reference directions.

PGAPack supports both GRGA and SSGA and variants in between via *parameterized* population replacement. For example, the `PGASet` calls

```
PGASetPopSize (ctx, 200);
PGASetNumReplaceValue (ctx, 10);
PGASetPopReplaceType (ctx, PGA_POPREPL_BEST);
```

specify that each generation a new population is created consisting of ten strings created via recombination, and the 190 most fit strings from the old population. The 190 strings can also be selected

randomly, with or without replacement, by setting the second argument of `PGASetPopReplaceType` to `PGA_POPREPL_RANDOM_REP` or `PGA_POPREPL_RANDOM_NOREP`, respectively.

For selecting restricted tournament replacement `PGA_POPREPL_RTR` is used. The default for the window size (number of members of the old population that are chosen for comparison with a new candidate) is $\min(n, N/20)$ where n is the string length and N is the population size [21]. The window size can be set or queried with `PGASetRTRWindowSize` and `PGARTRWindowSize`, respectively. Note that when restricted tournament replacement is in use, the maximum number of new candidates is limited with the number set with `PGASetNumReplaceValue` but fewer—depending on fitness—may be replaced into the new population. Note that it depends on the selection which individuals in the old population are replaced. Since restricted tournament replacement is an elitist strategy the overall fitness never diminishes with this replacement strategy.

For pairwise best replacement `PGA_POPREPL_PAIRWISE_BEST` is used as the replacement type. Like restricted tournament replacement it is an elitist strategy.

For NSGA-II replacement `PGA_POPREPL_NSII` is used. For NSGA-II replacement `PGA_POPREPL_NSIII` is used. The number of auxiliary evaluation function can be set with `PGASetNumAuxEval` and the number of constraints can be set with `PGASetNumConstraint`. If the difference between the two is > 0 (i.e. the number of objectives is > 1), these auxiliary evaluations are used for multi-objective optimization. Only the NSGA-II and NSGA-III replacement are possible with these settings (i.e. when the number of objectives is > 1).

The replacement types *pairwise best*, *restricted tournament replacement*, NSGA-II, and NSGA-III replacement have selection pressure in addition to providing a population replacement strategy. So these can be used if a selection scheme without selection pressure (a tournament strategy with only one participant in the tournament or linear selection) is used.

By default, the number of new strings created each generation is 10 percent of the population size (an SSGA population replacement strategy). A GRGA can be implemented by setting `PGASetNumReplaceValue` to the population size (the default population size is 100). Setting `PGASetNumReplaceValue` to one less than the population size will result in an elitist GRGA, where the most fit string is always copied to the new population (since `PGA_POPREPL_BEST` is the default population replacement strategy).

Traditionally, strings created through recombination first undergo crossover and then mutation. Some practitioners [4] have argued that these two operators should be separate. By default, PGAPack applies mutation only to strings that did *not* undergo crossover.

This is equivalent to setting `PGASetMixingType` to `PGA_MIX_MUTATE_OR_CROSS` which is also the default. To have strings undergo *both* crossover and mutation, one should set `PGASetMixingType` to `PGA_MIX_TRADITIONAL`. Note that there is also a mode that will not mutate strings that are not also crossed over. This can be enabled by setting `PGASetMixingType` to `PGA_MIX_MUTATE_AND_CROSS`.

If an evolutionary algorithm variant without crossover is used or if special crossover techniques with more than two parents should be applied, all the logic can be implemented in a custom crossover operator and the `PGASetMixingType` can be set to `PGA_MIX_MUTATE_ONLY`. In this mode no crossover is performed at all.

There is also a legacy interface that uses the functions `PGASetMutationOrCrossoverFlag`, `PGASetMutationAndCrossoverFlag` and `PGASetMutationOnlyFlag`. The legacy interface should not be used for new code.

By default, PGAPack allows duplicate strings in the population. Some practitioners advocate not allowing duplicate strings in the population in order to maintain diversity. The function call `PGASetNoDuplicatesFlag(ctx, PGA_TRUE)` will not allow duplicate strings in the population: It repeatedly applies the mutation operator (with an increasing mutation rate) to a duplicate string until it no longer matches any string in the new population. If the mutation rate exceeds 1.0, however, the duplicate string *will* be allowed in the population, and a warning message will be issued.

Figure 4.2 shows the generic population replacement scheme in PGAPack. Both populations k and $k + 1$ are of fixed size (the value returned by `PGAGetPopSize`). First, `PGAGetPopSize - PGASetNumReplaceValue` strings are copied over directly from generation k . The way the strings are chosen, the most fit, or randomly with or without replacement, depends on the value set by `PGASetPopReplaceType`. The remaining `PGASetNumReplaceValue` strings are created by crossover and mutation.

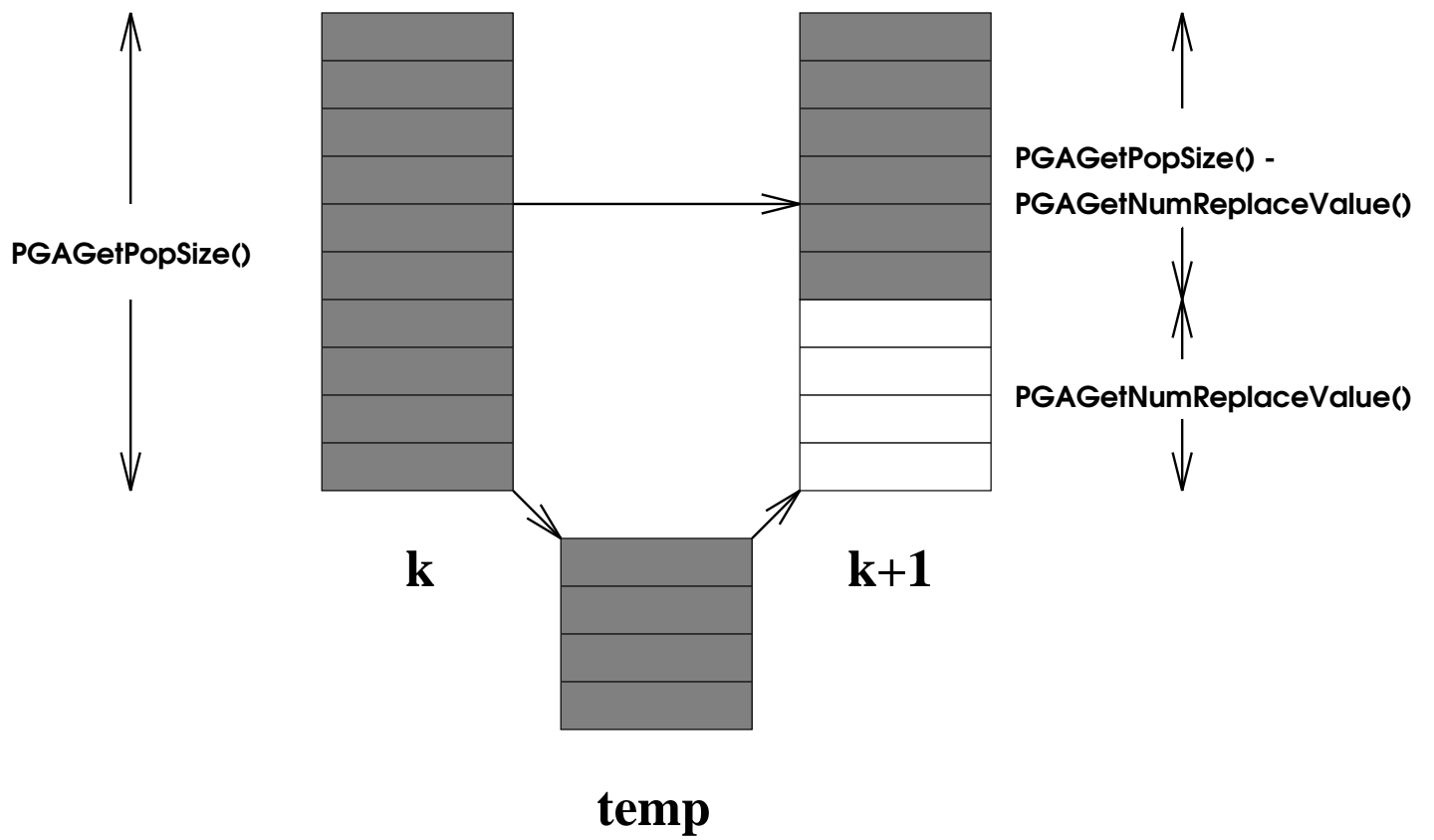


Figure 4.2: Population Replacement

4.3 Stopping Criteria

PGAPack terminates when at least one of the stopping rule(s) specified has been met. The three stopping rules are (1) iteration limit exceeded, (2) population too similar, and (3) no change in the best solution found in a given number of iterations. The default is to stop when the iteration limit (by default, 1000 iterations) is reached. Note that when ϵ -constraint optimization is in use, stopping is not triggered as long as $\epsilon > 0$, see section 4.9.

The choice of stopping rule is set by `PGASetStoppingRuleType`. For example, `PGASetStoppingRuleType(ctx, PGA_STOP_MAXITER)` is the default. Other choices are `PGA_STOP_TOOSIMILAR` and `PGA_STOP_NOCHANGE` for population too similar and no change in the best solution found, respectively. `PGASetStoppingRuleType` may be called more than once. The different stopping rules specified are *ored* together.

If `PGA_STOP_MAXITER` is one of the stopping rules, `PGASetMaxGAIterValue(ctx, 500)` will change the maximum iteration limit to 500. If `PGA_STOP_NOCHANGE` is one of the stopping rules, `PGASetMaxNoChangeValue(ctx, 50)` will change from 100 (the default) to 50 the maximum number of iterations in which no change in the best evaluation is allowed before the GA stops. If `PGA_STOP_TOOSIMILAR` is one of the stopping rules, `PGASetMaxSimilarityValue(ctx, 99)` will change from 95 to 99 the percentage of the population allowed to have the same evaluation function value before the GA stops.

4.4 Initialization

Strings are either initialized randomly (the default), or set to zero. The choice is specified by setting the second argument of `PGASetRandomInitFlag` to either `PGA_TRUE` or `PGA_FALSE`, respectively. Random initialization depends on the datatype.

If binary-valued strings are used, each gene is set to 1 or 0 with an equal probability. To set the probability of randomly setting a bit to 1 to 0.3, use `PGASetBinaryInitProb(ctx, 0.3)`.

For integer-valued strings, the default is to set the strings to a permutation on a range of integers. The default range is $[0, L - 1]$, where L is the string length. `PGASetIntegerInitPermute(ctx, 500, 599)` will set the permutation range to $[500, 599]$. The length of the range *must* be the same as the string length.

Alternatively, `PGASetIntegerInitRange` will set each gene to a random value selected uniformly from a specified range. For example, the code

```
stringlen = PGAGetStringLength(ctx);
for(i=0; i<stringlen; i++) {
    low[i] = 0;
    high[i] = i;
}
PGASetIntegerInitRange(ctx, low, high);
```

will select a value for gene i uniformly randomly from the interval $[0, i]$.

If real-valued strings are used, the alleles are set to a value selected uniformly randomly from a specified interval. The interval may be specified with either the `PGASetRealInitRange` or `PGASetRealInitPercent` functions. For example, the code

```
stringlen = PGAGetStringLength(ctx);
for(i=0; i<stringlen; i++) {
    low[i] = -10.0;
    high[i] = (double) i;
}
PGASetRealInitRange(ctx, low, high);
```

will select a value for allele i uniformly randomly from the interval $[-10.0, i]$. This is the default strategy for initializing real-valued strings. The default interval is $[0, 1.0]$.

`PGASetRealInitPercent` specifies the interval with a median value and percent offset. For example,

```
stringlen = PGAGetStringLength(ctx);
for(i=1; i<=stringlen; i++) {
```

```

    median [i-1] = (double) i;
    percent [i-1] = .5;
}
PGASetRealInitPercent(ctx, median, percent);

```

will select a value for allele i uniformly randomly from the increasing intervals $[\frac{1}{2}i, \frac{3}{2}i]$. Note that if the median value is zero for some i , then an interval of $[0, 0]$ will be defined.

If character-valued strings are used, `PGASetCharacterInitType(ctx, PGA_CINIT_UPPER)` will set the allele values to uppercase alphabetic characters chosen uniformly randomly. Other options are `PGA_CINIT_LOWER` for lower case letters only (the default) and `PGA_CINIT_MIXED` for mixed case letters, respectively.

4.5 Selection

The selection phase allocates reproductive trials to strings on the basis of their fitness. PGAPack supports five selection schemes: proportional selection, stochastic universal selection, truncation selection, tournament selection (default is binary tournament selection), and probabilistic binary tournament selection. A sixth scheme which is called *linear selection* that is not a selection scheme in the genetic sense (it has no selection pressure) is used for evolutionary algorithms that rely on modification of individuals that later replace their parent if the offspring has higher fitness, so the selection pressure is applied in the replacement strategy. The linear scheme is guaranteed to return individuals in population order.

The choice may be specified by setting the second argument of `PGASetSelectType` to one of `PGA_SELECT_PROPORTIONAL`, `PGA_SELECT_SUS`, `PGA_SELECT_TRUNCATION`, `PGA_SELECT_TOURNAMENT`, `PGA_SELECT_PTournament`, and `PGA_SELECT_LINEAR` for proportional, stochastic universal, truncation, tournament, probabilistic tournament selection, and linear selection, respectively. The default is tournament selection. For tournament selection, the size of the tournament (number of participants) can be set e.g., with `PGASetTournamentSize(ctx, 3)`. The default is binary tournament (size = 2). To allow a more fine-grained selection pressure, the tournament size is a floating-point value. The integer part of that value specifies the minimum tournament size. For each tournament for the fractional part, a biased coin is flipped (using `PGARandomFlip`) and the tournament size is increased by one if the outcome is positive. This mechanism for fine-grained tournaments was first proposed by Harik and Goldberg [18] and later rediscovered by Filipović et. al. [11].

Note that with a tournament size of 1 (or with the linear selection scheme) there is no selection pressure. Having no selection pressure in this step can be compensated by using a replacement scheme with selection pressure, i.e., one of restricted tournament replacement or pairwise best replacement, see section 4.2 for details on population replacement. If no selection pressure is used in the selection scheme *and* in the population replacement strategy, the genetic search degenerates to a random walk.

In addition, for tournament selection it can be specified if the selection is *with* or *without* replacement using the function `PGASetTournamentWithReplacement` with a parameter of `PGA_FALSE` or `PGA_TRUE`. Sampling without replacement guarantees that for n tournaments, each individual participates in the same number of tournaments (as long as n multiplied by the tournament size is a multiple of the population size) [14]. This was later re-invented by Sokolov and Whitley under the name *Unbiased Tournament Selection* [24].

The default sampling is *with* replacement as if `PGASetTournamentWithReplacement` had been called with the parameter `PGA_TRUE`. The probabilistic tournament selection is always binary (two participants in the tournament), the default probability that the string that wins the tournament is selected is 0.6. It may be set to 0.8, for example, with `PGASetPTournamentProb(ctx, 0.8)`. The tournament for probabilistic tournament selection is always with replacement. The truncation selection by default selects half of the population. This proportion of selected individuals can be set with `PGASetTruncationProportion` for which the default value is 0.5.

When using multi-objective optimization with, e.g., the NSGA-II [10] population replacement (see section 4.2), it is possible to either use a selection scheme with or without selection pressure. However, selection schemes that rely on direct comparison of individuals (e.g. tournament selection) will sort by the domination rank of the individuals established by the NSGA-II algorithm. This is because for multi-objective optimization there is no full order established by multiple objectives as would be the case for single-objective

optimization. This may result in less selection pressure because multiple individuals will typically have the same rank. This lower selection pressure is compensated by the selection pressure introduced by the NSGA-II (or -III) population replacement algorithm.

Most selection schemes (except stochastic universal selection) already return a randomized sequence. In previous implementations *all* sequences got an additional randomization step. By default this is no longer the case (except for `PGA_SELECT_SUS`). You can enable the previous behavior by setting it to `PGA_TRUE` with `PGASetRandomizeSelect`. Note that even with this flag set, the sequence returned by the linear selection scheme is never randomized. This has adverse effects on crossover with linear selection: With this scheme the same two adjacent population members are always crossed over which makes crossover almost ineffective. Linear selection is typically only useful when using special mutation operators most often with `PGASetMixingType` set to `PGA_MIX_MUTATE_ONLY`. If you need a randomized sequence without selection pressure, use tournament selection without replacement with a tournament size of one.

4.6 Crossover

The crossover operator takes bits from each parent string and combines them to create child strings. The type of crossover may be specified by setting `PGASetCrossoverType` to `PGA_CROSSOVER_ONEPT`, `PGA_CROSSOVER_TWOPT`, `PGA_CROSSOVER_UNIFORM`, or `PGA_CROSSOVER_SBX` for one-point, two-point, uniform, or simulated binary (SBX) crossover, respectively. The default is two-point crossover. By default the crossover rate is 0.85. It may be set to 0.6 by `PGASetCrossoverProb(ctx, 0.6)`, for example. Simulated binary crossover is available only for integer and real genes.

Uniform crossover and simulated binary crossover are parameterized by p_u , the probability of swapping two parent bit values [25] in the case of uniform crossover and for mutating an allele for SBX. By default, $p_u = 0.5$. The function call `PGASetUniformCrossoverProb(ctx, 0.7)` will set $p_u = 0.7$.

SBX uses a polynomial distribution with a parameter η_c that defines how far the child may deviate from each parent. For high values of this parameter, children stay nearer to the parents [6]. Recommended values of this parameter are typically in the range 2–5, the default is 2 and a different value can be set with `PGASetCrossoverSBXEta`.

When crossing strings with SBX, typically for each allele a new random number is computed for the polynomial distribution. With `PGASetCrossoverSBXOncePerString` you can define that a random number is only drawn once for each individual to be crossed over. This ensures that the child is on the line in N-dimensional space between the two parents *if all alleles are crossed over*. This may be beneficial when handling optimization problems that are not decomposable [23] similar to the crossover rate in differential evolution [22, p. 98].

Crossover types that may yield child individuals outside the range of the parents (currently only SBX) may want to call `PGASetCrossoverBoundedFlag` or `PGASetCrossoverBounceBackFlag` with the context variable and `PGA_TRUE` to select an algorithm that keeps the child alleles within the bounds of the initialization ranges of the gene for each allele. These parameters work analogous to `PGASetMutationBoundedFlag` and `PGASetMutationBounceBackFlag` for mutation. For the bounce-back implementation the parent *nearer* to the initialisation boundary is used for each check.

4.7 Mutation

The mutation *rate* is the probability that a gene will undergo mutation. The mutation rate is independent of the datatype used. The default mutation rate is the reciprocal of the string length. The function call `PGASetMutationProb(ctx, .001)` will set the mutation rate to .001.

The *type* of mutation depends on the data type. For binary-valued strings, mutation is a bit complement operation. For character-valued strings, mutation replaces one alphabetic character with another chosen uniformly randomly. The alphabetic characters will be lower, upper, or mixed case depending on how the strings were initialized.

For integer-valued strings, if the strings were initialized to a permutation and gene i is to be mutated, the default mutation operator swaps gene i with a randomly selected gene. If the strings were initialized to

a random value from a specified range and gene i is to be mutated, by default gene i will be replaced by a value selected uniformly random from the initialization range.

The mutation operator for integer-valued strings may be changed irrespective of how the strings were initialized. If `PGASetMutationType` is set to `PGA_MUTATION_RANGE`, gene i will be replaced with a value selected uniformly randomly from the initialization range. If the strings were initialized to a permutation, the minimum and maximum values of the permutation define the range. If `PGASetMutationType` is set to `PGA_MUTATION_PERMUTE`, gene i will be swapped with a randomly selected gene. If `PGASetMutationType` is set to `PGA_MUTATION_CONSTANT`, a constant integer value (by default one) will be added (subtracted) to (from) the existing allele value. The constant value may be set to 34, for example, with `PGASetMutationIntegerValue(ctx, 34)`.

Three of the real-valued mutation operators are of the form $v \leftarrow v \pm p \times v$, where v is the existing allele value. They vary by how p is selected. First, if `PGASetMutationType` is set to `PGA_MUTATION_CONSTANT`, p is the constant value 0.01. It may be set to .02, for example, with `PGASetMutationRealValue(ctx, .02)`. Second, if `PGASetMutationType` is set to `PGA_MUTATION_UNIFORM`, p is selected uniformly from the interval (0,1). To select p uniformly from the interval (0,1) set `PGASetMutationRealValue(ctx, 1)`. Third, if `PGASetMutationType` is set to `PGA_MUTATION_GAUSSIAN`, p is selected from a Gaussian distribution (this is the default real-valued mutation operator) with mean 0 and standard deviation 0.1. To select p from a Gaussian distribution with mean 0 and standard deviation 0.5 set `PGASetMutationRealValue(ctx, .5)`. Finally, if `PGASetMutationType` is set to `PGA_MUTATION_RANGE`, gene i will be replaced with a value selected uniformly random from the initialization range of that gene.

For integer and real genes there is a polynomial mutation operator selected with the mutation type constant `PGA_MUTATION_POLY` [8]. It works by drawing a random number from a polynomial probability density function for a fixed mutation interval. The mutation interval by default is between the current allele value and the lower/upper initialisation range of the gene. Unless you also call `PGASetMutationBoundedFlag` or `PGASetMutationBounceBackFlag` to keep mutation within the bounds of the initialization range, the default value does not make much sense (because the current value may already exceed the boundary). In that case (and other cases where you want a fixed mutation range) you can call `PGASetMutationPolyValue` to set the mutation range. The polynomial mutation distribution has a parameter η_m that specifies how likely values far away from the current allele are selected, the higher this value gets, the nearer the mutated values stays to the parent. You can set this parameter with `PGASetMutationPolyEta`, the default is 100 [7].

For Differential Evolution (DE), the strategy is implemented as the mutation type `PGA_MUTATION_DE`. Note that for the full DE algorithm not just a special mutation type is needed, see section 2.4 for an introduction. You typically want to chose mutation only, linear selection, and pairwise-best replacement. For DE, real-valued genes are typically called *vectors* (because a gene is a vector of real-valued alleles), we use that term in the following.

The default strategy of DE is to compute the distance of a pair of random vectors in the population and add this difference to a randomly-chosen third vector:

$$V_{i,g} = X_{r_0,g} + F \cdot (X_{r_1,g} - X_{r_2,g}) \quad (4.1)$$

Where g denotes the generation, i is the running population index, and r_0 , r_1 , and r_2 are random population indeces different from each other and the running index i . The factor F , called the scale factor, is a parameter of the search and can be specified with `PGASetDEScaleFactor`, the default is 0.9. The range for this parameter is $[0, 2]$ but typical values are in the range $[0.3, 1)$, the exact value 1.0 should not be chosen because it reduces the number of mutants and thus potentially the genetic variance [22, p. 75].

The resulting mutant vector V_i , also called the *donor* vector, is then combined via crossover with the i^{th} member of the population X_i . Note that the crossover in this implementation of DE is *not* the normal genetic algorithm crossover (from section 4.6), in fact when using DE, crossover is typically turned off by setting `PGASetMixingType` to `PGA_MIX_MUTATE_ONLY`. Instead DE uses its own crossover implementation within the `PGA_MUTATION_DE` mutation implementation. For each vector element (allele), a random variable with a crossover probability specified by `PGASetDECrossoverProb` (default 0.9) is chosen. In the default binomial crossover variant of DE this variable selects an element from either the donor vector V_i or X_i . In the DE literature this parameter is typically called Cr . At least one random element from V_i is always selected, so with a crossover probability of 0, exactly one element from the donor vector is selected. With a crossover probability of 1, *all* elements from the donor vector are selected. Crossover plays a significant

role in optimisation. For decomposable problems (where each dimension of the problem can be optimized separately [23]) a low crossover rate $0 \leq Cr \leq 0.2$ is a good choice. For non-decomposable problem a high crossover rate should be chosen, i.e. $0.9 \leq Cr \leq 1$ [22, p. 98].

The resulting individual after crossover is placed in the new population. If pairwise-best population replacement is selected (the default in the DE literature, see section 4.2) it is later compared with the old individual X_i in the old population which it replaces if it has better fitness.

There are different variants of DE and a notation was established to distinguish these variants. The notation uses a 4-tuple where each element is delimited with a '/'. The first element in the tuple is always the string *DE* for Differential Evolution. The second element describes the variant. The most common (often called the *classic* variant) selects a random element from the population for modification and is therefore called *rand*, see equation 4.1. The third tuple-element is the number of difference-pairs applied to the modified element, it is typically one or two. If we use two differences, the formula in equation 4.1 would become:

$$V_{i,g} = X_{r0,g} + F \cdot (X_{r1,g} - X_{r2,g}) + F \cdot (X_{r3,g} - X_{r4,g}) \quad (4.2)$$

The number of differences can be specified with `PGASetDNumDiffs` and defaults to 1. Note that not all DE strategies use this parameter.

The fourth final element specifies the crossover strategy. For the default DE-crossover strategy following a binomial distribution (in standard GA terms this type of crossover is often called *uniform* crossover) this final element is termed *bin*. The crossover for DE can be set with `PGASetDECrossoverProb`. The default is binomial crossover `PGA_DE_CROSSOVER_BIN`, exponential crossover can be set with `PGA_DE_CROSSOVER_EXP`. So for the default DE strategy the name is *DE/rand/1/bin* with the exponential crossover selected, we would get *DE/rand/1/exp*. Binomial crossover tosses a biased coin with probability *Cr* for each allele. If the coin turns out '1', the allele is taken from the donor vector, otherwise the allele from the current individual is retained. For exponential crossover an index in the gene is randomly selected and taken from the donor vector. For all subsequent alleles (starting at the randomly selected index) a coin is tossed. As long as the coin is '1', the allele is taken from the donor vector. The first time the coin-toss doesn't yield a '1', all the remaining alleles are taken retained from the original individual. This is a form of two-point crossover like in other types of genetic algorithms but with a different distribution. The binomial crossover is beneficial if the allele positions in the problem are uncorrelated. If there is a correlation between allele positions, exponential crossover may be beneficial [31].

The DE variant (the second tuple element) can be selected with `PGASetDEVariant` and defaults to `PGA_DE_VARIANT_RAND`. Another variant is the *best* variant which uses the current best individual for modification according to equation 4.3. This variant is selected with the constant `PGA_DE_VARIANT_BEST`.

$$V_{i,g} = X_{\text{best},g} + F \cdot (X_{r1,g} - X_{r2,g}) \quad (4.3)$$

A third variant called *either-or* [22, p. 117] is selected with the constant `PGA_DE_VARIANT_EITHER_OR`. It randomly selects among a mutation operator and a recombination operator according to equation 4.4.

$$V_{i,g} = \begin{cases} X_{r0,g} + F \cdot (X_{r1,g} - X_{r2,g}) & \text{if } \text{rand}_i(0, 1) < p_F \\ X_{r0,g} + K \cdot (X_{r1,g} + X_{r2,g} - 1 \cdot X_{r0,g}) & \text{otherwise} \end{cases} \quad (4.4)$$

The probability p_F defaults to 0.5 and can be set with `PGASetDEProbabilityEO`. The parameter K defaults to $0.5 \cdot (F + 1)$ [22, p. 118] where F is the scale factor, it can be set with `PGASetDEAuxFactor`. Note that the either-or variant ignores the parameter specifying the number of difference vectors (specified with `PGASetDNumDiffs`). The expression $\text{rand}_i(0, 1)$ denotes a random number in the range $[0, 1)$ which is re-generated for each individual i .

When computing the donor vector V_i , the scale factor F can be perturbed. This can either be done anew for each individual or for each *allele* of each individual. The first variant is called *dither* while the second variant is called *jitter*. Note that jitter not only changes the length of the difference vector but also its orientation [22, p. 80]. With uniform jitter we get a new factor F_j , the index j denoting the allele while i is the index of the current individual [22, p. 80]:

$$F_j = (F + K_{\text{jit}} \cdot (\text{rand}_j(0, 1) - 0.5)), \quad K_{\text{jit}} < 2 \cdot F \quad (4.5)$$

The same applies for dither, but in the case of dither the factor is either applied anew for each individual (when setting `PGASetDEDitherPerIndividual` to `PGA_TRUE`) or only once per generation (the default being once per generation), note that for the case where the dither is applied once per generation the index i of K_{jit} in equation 4.6 would refer to the generation:

$$F_i = (F + K_{dit} \cdot (\text{rand}_i(0, 1) - 0.5)), \quad K_{dit} < 2 \cdot F \quad (4.6)$$

The new F_j and/or F_i replaces F in equations 4.1–4.4 and both, the default K_{jit} and the default K_{dit} are zero by default. If both are non-zero, both are applied. Defaults can be set with `PGASetDEJitter` and `PGASetDEDither`, respectively. Very small amounts of uniformly distributed jitter (on the order of 0.001) have been recommended for some problems [22, p. 90] like digital filter design [22, p. 440]. Likewise quite large amounts of dither (on the order of 0.5) are recommended for these problems.

Some of the integer- and real-valued mutation operators may generate allele values outside the initialization range of that gene. By default, the allele value will *not* be reset to the lower (upper) value of the initialization range for that gene. By setting `PGASetMutationBoundedFlag(ctx, PGA_TRUE)` the allele values will be set to the value of the bound if they fall outside of the initialization range. It was argued that setting the value to the bound would reduce the genetic variance and could lead to premature convergence if several individuals get the same value [22, p. 204]. Therefore an alternative called “Bounce-Back” was proposed: If `PGASetMutationBounceBackFlag(ctx, PGA_TRUE)` is set, the new value is set to a random value between the old value and the bound.

A note on the use of the DE mutation type together with other selection or replacements schemes: The DE mutation is very disruptive. It will not work well or not work at all with a non-elitist replacement scheme. Due to the high disruption, if not retaining at least one best individual in each generation, it is very likely that the search will diverge. Good choices for an elitist strategy are the two elitist replacement schemes `PGA_POPREPL_PAIRWISE_BEST` (which is the standard replacement scheme for Differential Evolution) and `PGA_POPREPL_RTR`. The latter may help if the algorithm stagnates due to premature convergence. In that case RTR can help to retain more genetic diversity. For details see section 4.2.

4.8 Restart

The restart operator reseeds a population from the best string. It does so by seeding the new population with the best string and generating the remainder of the population as mutated variants of the best string.

By default the restart operator is not invoked. Setting `PGASetRestartFlag(ctx, PGA_TRUE)` will cause the restart operator to be invoked. By default PGAPack will restart every 50 iterations. `PGASetRestartFrequencyValue(ctx, 100)` will restart every 100 iterations instead. When creating the new strings from the best string an individual allele undergoes mutation with probability 0.5. This can be changed to 0.9 with the function call `PGASetRestartAlleleChangeProb(ctx, 0.9)`.

For binary-valued strings the bits are complemented. For integer- and real-valued strings the amount to change is set with `PGASetMutationIntegerValue` and `PGASetMutationRealValue`, respectively. Character-valued strings are changed according to the rules in Section 4.7 for mutating character strings.

4.9 String Evaluation and Fitness

In a genetic algorithm each string is assigned a nonnegative, real-valued *fitness*. This is a measure, relative to the rest of the population, of how well that string satisfies a problem-specific metric. In PGAPack calculating a string’s fitness is a two-step process. First, the *user* supplies a real-valued evaluation (sometimes called the raw fitness) of each string. Second, this value is mapped to a fitness value.

It is the user’s responsibility to supply a function to evaluate an individual string. As discussed in Section 4.1, the name of this function is specified as the second argument to `PGARun`. The calling sequence for this function (which we call `evaluate` in the rest of this section, but may have any name) *must* follow the format given here. In C the format is

```
double evaluate (PGAContext *ctx, int p, int pop, double *aux);
```

and in Fortran

```
double precision function evaluate (ctx, p, pop, aux)
integer ctx, p, pop
double precision aux(*)
```

The function `evaluate` will be called by `PGARun` whenever a string evaluation is required. `p` is the index of the string in population `pop` that will be evaluated. The correct values of `p` and `pop` will be passed to the evaluation function by `PGARun`. (If `PGARun` is not used, `PGAEvaluate` must be. See Chapter 5.) As shown below, `p` and `pop` are used for reading (and sometimes writing) allele values. Sample evaluation functions are shown in Figures 2.1 and 2.2, and online in the `./examples` directory.

In addition to returning just one evaluation, `PGAPack` supports additional *auxiliary* evaluations. The default use for this mechanism is the specification of *constraints* on the objective function. If a problem does not allow all areas of the search space because it may contain invalid solutions, additional restrictions on the validity of points in the search space may be specified via constraints.

Another use-case for auxiliary evaluations is multiobjective optimization: The algorithm is not just searching for *one* solution but for an array of objectives that can usually not all be optimized to their optimum value. Instead a better value for one objective may necessitate a worse value for another objective. A multiobjective algorithm tries to find many *non dominated* solutions to a problem (a solution is said to dominate another solution if it is better in at least one objective but not worse in any other objective). These non-dominated solutions are said to lie on a *Pareto Front* after the mathematician Vilfredo Pareto who first defined the concept today known as *pareto optimality*.

`PGAPack` now implements multi-objective optimization with the Nondominated Sorting Genetic Algorithm (Version 2) [10] as a replacement strategy. You can mix multi-objective optimization and constraints. See below and section 4.2 for details.

By default, auxiliary evaluations are used for constraints. All auxiliary evaluation are summed if the value is positive. If it is zero or negative, the constraint is not violated and not included in the sum. So the algorithm is minimizing constraint violations. Individuals are sorted by the amount of their constraint violations and the value of the objective function. If an individual without constraint violations is compared to one with constraint violations, the one without violations wins. For two individuals with constraint violations the one with the lower sum of violations wins. For two individuals without constraint violations the normal comparison (depending on the direction of the search, i.e. minimization or maximization) is used. This algorithm works better than trying to code the constraint violations into a complicated evaluation function. It was shown to work better than customized penalty functions by Deb [5].

With this algorithm for optimizing constraints, the constraints are optimized first, only after solutions without constraint violations have been found is the objective function considered. This has the drawback that for certain problems the search will end up in a region of the search space where the constraints are not violated but where no good solutions exist. So with some problems the solutions found are very far from the optimum. An idea by Takahama and Sakai [30], [29] introduces an ϵ -constraint mechanism. An epsilon tolerance is introduced and initialized with the constraint violation sum of the θ -best individual. The index of the individual θ can be set with `PGASetEpsilonTheta`, the default is 0.2 the population size.

The comparison of evaluations is modified to include an ϵ -tolerance: If both individuals have a constraint violation below the tolerance, the evaluation is compared. If only one individual exceeds the tolerance the other individual wins and if both exceed the tolerance, the one with the lower constraint sum wins. The ϵ is slowly decreased until it becomes zero at some generation T_c . The slope of decrease can be specified with the `PGASetEpsilonExponent` function, values between 2 (slow decrease) and 10 (fast decrease) are possible. The default is an adaptive algorithm for decrease of ϵ described in the 2010 paper [29] that works well in practise.

For using the ϵ -constraint method, the generation T_c until which ϵ is decreased needs to be set using the `PGASetEpsilonGeneration` function, the default is zero. Note that `PGASetEpsilonGeneration` needs to be below the value set with `PGASetMaxGAIterValue` even if the latter is not used as a stopping criterion. Also note that the stopping criteria (see section 4.3) are modified to not stop as long as ϵ is not zero.

Auxiliary evaluations are returned in an array pointed to by the `aux` parameter of the users's evaluation function. To use auxiliary evaluations, the number of auxiliary evaluations has to be specified with the `PGASetNumAuxEval` function which gets the number of auxiliary evaluations as the second parameter. The

default is 0. If you want to use multi-objective optimization, optionally with constraints, you need to specify the number of constraints using the `PGASetNumConstraint` function. By default the number of constraints is equal to the number of auxiliary evaluations. So if you want to use multi-objective evaluation you need to set the number of constraints lower (optionally to zero if you have only multi-objective optimization without constraints) than the number of auxiliary evaluations.

Note that auxiliary evaluations can not be used together with selection schemes that use mechanisms where individuals are not directly compared. These currently are proportional selection and stochastic universal selection, see section 4.5.

Traditionally, genetic algorithms assume fitness values are nonnegative and monotonically increasing the more fit a string is. The user's evaluation of a string, however, may reflect a minimization problem and/or be negative. Most modern selection algorithms (e.g. the default tournament variants) directly compare individuals and will directly use the user's evaluation. There are two selection mechanisms, `PGA_SELECT_SUS` and `PGA_SELECT_PROPORTIONAL` which need a nonnegative and monotonically increasing fitness. *Only for these the user's evaluation value is mapped to a nonnegative and monotonically increasing fitness value.*

You may think of the algorithm used as follows (actually for the ranking method the evaluation value is never translated): First, all evaluations are mapped to positive values (if any were negative). Next, these values are translated to a maximization problem (if the direction of optimization specified was minimization). Finally, these values are mapped to a fitness value by using the identity (the default), linear ranking, or linear normalization. The choice of fitness mapping may be set with the function `PGASetFitnessType`. The second argument must be one of `PGA_FITNESS_RAW`, `PGA_FITNESS_RANKING`, or `PGA_FITNESS_NORMAL`, for the identity, linear ranking, or linear normalization, respectively.

Note that `PGA_FITNESS_RAW` and `PGA_FITNESS_NORMAL` are subject to overflows if you have very large (or very small negative) fitness values. If this occurs, an error message is printed and the program terminates. Letting the search continue with such an overflow would map many *different* evaluation values to the *same* fitness. For such ill-conditioned problems you should use the ranking variant `PGA_FITNESS_RANKING`.

A *linear rank* fitness function [1, 32] is given by

$$Min + (Max - Min) \cdot \frac{\text{rank}(\mathbf{p}) - 1}{N - 1}, \quad (4.7)$$

where $\text{rank}(\mathbf{p})$ is the index of string \mathbf{p} in a list sorted in order of decreasing evaluation function value, and N is the population size. Ranking requires that $1 \leq Max \leq 2$, and $Min + Max = 2$. The default value for Max is 1.2. It may be set to 1.1 with `PGASetMaxFitnessRank(ctx, 1.1)`.

In *linear normalization* the fitness function is given by

$$K - (\text{rank}(\mathbf{p}) \cdot C), \quad (4.8)$$

where K and C are the constants $\sigma \cdot N$ and σ , where σ is the standard deviation of the user's evaluation function values after they have been transformed to positive values for a maximization problem.

If the direction of optimization is minimization, the values are remapped for maximization. The function call `PGASetFitnessMinType(ctx, PGA_FITNESSMIN_CMAX)` will remap by subtracting the worst evaluation value from each evaluation value (this is the default). The worst evaluation value is multiplied by 1.01 before the subtraction so that the worst string has a nonzero fitness. The function call `PGASetFitnessCmaxValue(ctx, 1.2)` will change the multiplier to 1.2. Alternatively, if `PGA_FITNESSMIN_RECIPROCAL` is specified the remapping is done by using the reciprocal of the evaluation function.

Note that for algorithms that can directly compare individuals in the selection method (any of the tournament selection methods, truncation selection, and linear selection, see section 4.5) or in the replacement scheme (restricted tournament replacement or pairwise best replacement, see section 4.2) do not use the fitness but compare the evaluation value (and optionally the sum of constraint violations) directly.

4.10 Accessing Allele Values

For each of the native data types, PGAPack provides a matched pair of functions that allow the user to read or write (change) any allele value. If the data type is `PGA_DATATYPE_BINARY`

```
int bit;
bit = PGAGetBinaryAllele (ctx, p, pop, i);
```

will assign to `bit` the binary value of the `i`th gene in string `p` in population `pop`. To set the `i`th gene in string `p` in population `pop` to 1, use

```
PGASetBinaryAllele(ctx, p, pop, i, 1);
```

If the data type is `PGA_DATATYPE_INTEGER`

```
int k;
k = PGAGetIntegerAllele (ctx, p, pop, i);
```

will assign to `k` the integer value of the `i`th gene in string `p` in population `pop`. To set the `i`th gene in string `p` in population `pop` to 34, use

```
PGASetIntegerAllele(ctx, p, pop, i, 1, 34);
```

If the data type is `PGA_DATATYPE_REAL`

```
double x;
x = PGAGetRealAllele (ctx, p, pop, i);
```

will assign to `x` the real value of the `i`th gene in string `p` in population `pop`. To set the `i`th gene in string `p` in population `pop` to 123.456, use

```
PGASetRealAllele(ctx, p, pop, i, 1, 123.456);
```

If the data type is `PGA_DATATYPE_CHARACTER`

```
char c;
c = PGAGetCharacterAllele (ctx, p, pop, i);
```

will assign to `c` the character value of the `i`th gene in string `p` in population `pop`. To set the `i`th gene in string `p` in population `pop` to “Z”, use

```
PGASetCharacterAllele(ctx, p, pop, i, 1, 'Z');
```

4.10.1 Representing an Integer with a Binary String

A binary string may be used to represent an integer by *decoding* the bits into an integer value. In a binary coded decimal (BCD) representation, a binary string is decoded into an integer $k \in [0, 2^N - 1]$ according to

$$k = \sum_{i=1}^N b_i 2^{i-1}, \quad (4.9)$$

where N is the string length, and b_i the value of the i th bit. For example, to decode the integer `k` from the ten bits in bit positions 20–29, use

```
int k;
k = PGAGetIntegerFromBinary(ctx,p,pop,20,29);
```

The function `PGAEncodeIntegerAsBinary` will encode an integer as a binary string. For example, to encode the integer 564 as a 12-bit binary string² in the substring defined by bits 12–23, use

```
PGAEncodeIntegerAsBinary(ctx,p,pop, 12, 23, 564);
```

²Even though only ten bits are necessary to encode 564, the user may want to allow the GA any value between $[0, 4095]$, hence the twelve bits.

Table 4.1: Binary and Gray Codes

k	Eq. (4.9)	Gray code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

In a BCD representation, two numbers that are contiguous in their decimal representations may be far from each other in their binary representations. For example, 7 and 8 are consecutive integers, yet their 4-bit binary representations, 0111 and 1000, differ in the maximum number of bit positions.³ *Gray codes* define a different mapping of binary strings to integer values from that given by Eq. (4.9) and may alternatively be used for representing integer (or real, see below) values in a binary string. The second and third columns in Table 4.1 show how the integers 0–7 are mapped to Eq. (4.9) and to the *binary reflected* Gray code (the most commonly used Gray code sequence), respectively. In the binary reflected Gray code sequence, the binary representations of consecutive integers differ by only one bit (a Hamming distance of one).

To decode the integer k from a binary reflected Gray code interpretation of the binary string, use

```
k = PGAGetIntegerFromGrayCode(ctx,p,pop,20,29);
```

To encode 564 as a 12-bit binary string in the substring defined by bits 12–23 using a Gray code, use

```
PGAEncodeIntegerAsGrayCode(ctx,p,pop, 12, 23, 564);
```

4.10.2 Representing a Real Value with a Binary String

A binary string may also be used to represent a real value. The decoding of a binary string to a real-value is a two-step process. First, the binary string is decoded into an integer as described in Section 4.10.1. Next, the integer is mapped from the discrete interval $[0, 2^N - 1]$ to the real interval $[L, U]$ by using the formula

$$x = (k - a) \times (U - L) / (b - a) + L$$

(and generalizing $[0, 2^N - 1]$ to $[a, b]$). For example, to decode the `double` x from the 20 bits given by the binary string stored in bit positions 10–29 onto the interval $[-10.0, 20.0]$, use

```
x = PGAGetRealFromBinary(ctx,p,pop,10,29,-10.0,20.0);
```

To encode -18.3 on the interval $[-50.0, 50.0]$ using a 20-bit BCD binary string, use

```
PGAEncodeRealAsBinary(ctx,p,pop,0,19,-50.0,50.0,-18.3);
```

The functions `PGAGetRealFromGrayCode` and `PGAEncodeRealAsGrayCode` provide similar functionality for Gray-coded strings.

4.10.3 Example

As an example, suppose the user has a real-valued function f of three real variables x_1 , x_2 , and x_3 . Further, the variables are constrained as follows.

$$-10 \leq x_1 \leq 0$$

$$0 \leq x_2 \leq 10$$

³Technically, this is known as a Hamming cliff.

$$-10 \leq x_3 \leq 10$$

The user wishes to use 10 bits for the binary representation of x_1 and x_2 , and 20 bits for the binary representation of x_3 (perhaps for higher accuracy), and a Gray code encoding. This may be done as follows.

```
#include "pgapack.h"
double grayfunc (PGAContext *ctx, int p, int pop);
double f         (double x1, double x2, double x3);
int main(int argc, char **argv)
{
    PGAContext *ctx;
    ctx = PGACreate (&argc, argv, PGA_DATATYPE_BINARY, 40, PGA_MINIMIZE);
    PGASetUp      (ctx                                     );
    PGARun        (ctx, grayfunc                           );
    PGADestroy    (ctx                                     );
    return;
}

double grayfunc (PGAContext *ctx, int p, int pop)
{
    double x1, x2, x3, v;
    x1 = PGAGetRealFromGrayCode (ctx, p, pop, 0, 9, -10., 0.);
    x2 = PGAGetRealFromGrayCode (ctx, p, pop, 10, 19, 0., 10.);
    x3 = PGAGetRealFromGrayCode (ctx, p, pop, 20, 39, -10., 10.);
    v = f(x1,x2,x3);
    return(v);
}
```

In Fortran, the bit indices would be 1–10, 11–20, and 21–40, respectively. The number of bits allocated for the binary representation determines the accuracy with which the real value can be calculated. Note in this example the function `f` *need not be modified*; the function `grayfunc` is used as a “wrapper” to get variable values out of the GA and return the value calculated by `f`.

4.11 Report Options

`PGASetPrintFrequencyValue(ctx,40)` will print population statistics every 40 iterations. The default is every ten iterations. The best evaluation is *always* printed. To print additional statistics, set the second argument of the function `PGASetPrintOptions` to `PGA_REPORT_ONLINE`, `PGA_REPORT_OFFLINE`, `PGA_REPORT_WORST`, `PGA_REPORT_AVERAGE`, `PGA_REPORT_HAMMING`, or `PGA_REPORT_STRING` to print the online analysis, offline analysis, worst evaluation, average evaluation, Hamming distance, or string itself, respectively. `PGASetPrintOptions` may be called multiple times to specify multiple print options.

4.12 Utility Functions

4.12.1 Random Numbers

By default, PGAPack will seed its random number generator by using a value from the system clock. Therefore, each time PGAPack is run, a unique sequence of random numbers will be used. For debugging or reproducibility purposes, however, the user may wish to use the same sequence of random numbers each time. This may be done using the function `PGASetRandomSeed` to initialize the random number generator with the same seed each time, for example, `PGASetRandomSeed(ctx,1)`.

`PGARandom01(ctx,0)` will return a random number generated uniformly on $[0,1]$. If the second argument is not 0, it will be used to reseed the random number sequence. `PGARandomFlip` flips a biased coin. For example, `PGARandomFlip(ctx,.7)` will return `PGA_TRUE` approximately 70% of the time. `PGARandomInterval(-10,30)` will return an integer value generated uniformly on $[-10,30]$.

`PGARandomUniform (ctx,-50.,50.)` will return a real value generated uniformly randomly on the interval `[-50,50]`. `PGARandomGaussian (ctx,0.,1.)` will return a real value generated from a Gaussian distribution with mean zero and standard deviation one.

4.12.2 Print Functions

`PGAPrintPopulation(ctx,stdout,pop)` will print the evaluation function value, fitness value, and string for each member of population `pop` to `stdout`. This function may not be called until *after* `PGASetUp` has been called. `PGAPrintContextVariable(ctx,stdout)` will print the value of all fields in the context variable to `stdout`. `PGAPrintIndividual(ctx,stdout,p,pop)` will print the evaluation function value, fitness value, and string of individual `p` in population `pop` to `stdout`. `PGAPrintString(ctx,stdout,p,pop)` will print the string of individual `p` in population `pop` to `stdout`. `PGAPrintVersionNumber(ctx)` will print the PGAPack version number.

4.12.3 Miscellaneous

`PGAGetGAIterValue(ctx)` will return the current iteration of the GA. `PGAGetBestIndex(ctx,pop)` (`PGAGetWorstIndex`) will return the index of the most (least) fit member of population `pop`.

`PGAUpdateOffline(ctx,pop)` (`PGAUpdateOnline`) will update the offline (online) analysis based on the new generation's results. `PGAHammingDistance(ctx,pop)` returns a `double`, which is the average Hamming distance between the *binary* strings in population `pop`. The function call

```
PGAError(ctx, "popindex=", PGA_FATAL, PGA_INT, (void *)&popindex)
```

will print the message "popindex=-1" (assuming the value of `popindex` is -1) and then exit PGAPack. If the third argument had been `PGA_WARNING` instead, execution would have continued. In addition to `PGA_INT`, valid data types are `PGA_DOUBLE`, `PGA_CHAR`, and `PGA_VOID`.

4.13 Command-Line Arguments

PGAPack provides several command-line arguments. These are only available to C programs, although in some cases both C and Fortran programs can achieve the equivalent functionality with function calls. For example, `PGAUsage(ctx)` provides the same functionality as the `-pgahelp` command line option. See Chapter 11 for the function call equivalents.

<code>-pgahelp</code>	get this message
<code>-pgahelp debug</code>	list of debug options
<code>-pgadbg <level></code>	set debug option
<code>-pgadebug <level></code>	set debug option
<code>-pgaversion</code>	Print current PGAPack version number, parallel or sequential, and debug or optimized

Chapter 5

Explicit Usage

This chapter discusses how the user may obtain greater control over the steps of the GA by *not* using the `PGARun` command, but instead calling the data-structure-neutral functions directly. One ramification of this is that the `PGARun` interface no longer masks some of the differences between parallel and sequential execution. The examples in this chapter are written for sequential execution *only*. Chapter 9 shows how they may be executed in parallel.

5.1 Notation

To understand the calling sequences of the functions discussed in this chapter, one must know of the *existence* of certain data structures and the user interface for accessing them. It is *not* necessary to know how these data structures are implemented, since that is hidden by the user interface to `PGAPack`.

`PGAPack` maintains two populations: an *old* one and a *new* one. The size of each population is the value returned by `PGAGetPopSize`. In addition, each population contains two temporary working locations. The string length is the value specified to `PGACreate` and returned by `PGAGetStringLength`.

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 (i, p, pop) . In `PGAPack`, `pop` *must* be one of the two symbolic constants `PGA_OLDDPOP` or `PGA_NEWPOP` to refer to the old or new population, respectively. At the end of each GA iteration, the function `PGAUpdateGeneration` makes sure these symbolic constants are remapped to the correct population. The string index p must be either an integer between 0 and $P - 1$ (or 1 and P in Fortran) or one of the symbolic constants `PGA_TEMP1` or `PGA_TEMP2`, to reference one of the two temporary locations, respectively.

5.2 Simple Sequential Example

The example in Figure 5.1 is a complete `PGAPack` program that does *not* use `PGARun`. It is an alternative way to write the main program for the Maxbit example of Section 2.1. We refer to it as a simple example because it uses `PGARunMutationAndCrossover` to encapsulate the recombination step. The `PGACreate` and `PGASetUp` functions were discussed in the last chapter. `PGASetUp` creates and randomly initializes the initial population. This population, referred to initially by the symbolic constant `PGA_OLDDPOP`, is evaluated by the `PGAEvaluate` function. The third argument to `PGAEvaluate` is the name of the user's evaluation function. The function prototype for `evaluate` must be as shown in Figure 5.1 and discussed earlier in Sections 4.1 and 4.9. The `PGAFitness` function maps the user's evaluation function values into fitness values.

The `while` loop runs the genetic algorithm. `PGADone` returns `PGA_TRUE` if any of the specified stopping criteria have been met, otherwise `PGA_FALSE`. `PGASelect` performs selection on population `PGA_OLDDPOP`. `PGARunMutationAndCrossover` uses the selected strings to create the new population by applying the crossover and mutation operators. `PGAEvaluate` and `PGAFitness` evaluate and map to fitness values the newly created population. `PGAUpdateGeneration` updates the GA iteration count and resets several important internal arrays (don't forget to call it!). `PGAPrintReport` writes out genetic algorithm statistics

```

#include "pgapack.h"
double evaluate (PGAContext *ctx, int p, int pop, double *aux);

int main(int argc, char **argv)
{
    PGAContext *ctx;

    ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
    PGASetUp      (ctx);
    PGAEvaluate(ctx, PGA_OLDPOP, evaluate, NULL);
    PGAFitness (ctx, PGA_OLDPOP);

    while(!PGADone(ctx, NULL)) {
        PGASelect                (ctx, PGA_OLDPOP);
        PGARunMutationAndCrossover(ctx, PGA_OLDPOP, PGA_NEWPOP);
        PGAEvaluate              (ctx, PGA_NEWPOP, evaluate, NULL);
        PGAFitness               (ctx, PGA_NEWPOP);
        PGAUpdateGeneration      (ctx, NULL);
        PGAPrintReport           (ctx, stdout, PGA_OLDPOP);
    }
    PGADestroy(ctx);
    return(0);
}

```

Figure 5.1: Simple Example of Explicit Usage

according to the report options specified. Note that the argument to `PGAPrintReport` is the old population, since after `PGAUpdateGeneration` is called, the newly created population is in `PGA_OLDPOP`. Finally, `PGADestroy` releases any memory allocated by `PGAPack` when execution is complete.

The functions `PGADone`, `PGAUpdateGeneration`, and `PGAEvaluate` take an MPI communicator (see Appendix C and Chapter 9) as an argument. For *sequential* execution the value `NULL` should be specified for this argument. A parallel, or sequential *and* parallel, version of this example is given in Section 9.2.

5.3 Complex Example

The primary difference between the “complex” example in Figure 5.2 and the “simple” example in Figure 5.1 is that the steps encapsulated by `PGARunMutationAndCrossover` have been written out explicitly. The function `PGASortPop` sorts a population according to the criteria specified by `PGASetPopReplaceType` (Section 4.2). The sorted indices are accessed via `PGAGetSortedPopIndex`. In the example, the five lines that follow `PGASortPop` copy the strings that are not created by recombination from the old population to the new population.

The `while` loop that follows creates the remainder of the new population. `PGASelectNextIndex` returns the indices of the strings selected by `PGASelect`. `PGARandomFlip` flips a coin biased by the crossover probability to determine whether the selected strings should undergo crossover and mutation or should be copied directly into the new population. `PGACrossover` uses the parent strings `m1` and `m2` from population `PGA_OLDPOP` to create two child strings in the temporary locations `PGA_TEMP1` and `PGA_TEMP2` in `PGA_NEWPOP` population.

`PGAMutate` mutates the child strings and `PGACopyIndividual`, then copies them into the new population. If the strings do not undergo crossover and mutation, they are copied into the new population unchanged. The rest of the steps are the same as those in Figure 5.1, *except* that for illustrative purposes we call `PGAPrintReport` *before* `PGAUpdateGeneration`. In that case we use population `PGA_NEWPOP` as the population pointer.

```

#include "pgapack.h"
double evaluate(PGAContext *ctx, int p, int pop, double *aux);

int main(int argc, char **argv)
{
    PGAContext *ctx;
    int i, j, n, m1, m2, popsize, numreplace;
    double probcross;

    ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
    PGASetUp(ctx);
    probcross = PGAGetCrossoverProb(ctx);
    popsize = PGAGetPopSize(ctx);
    numreplace = PGAGetNumReplaceValue(ctx);
    PGAEvaluate(ctx, PGA_OLDPOP, evaluate, NULL);
    PGAFitness (ctx, PGA_OLDPOP );

    while(!PGADone(ctx, NULL)) {
        PGASelect (ctx, PGA_OLDPOP);
        PGASortPop(ctx, PGA_OLDPOP);
        n = popsize - numreplace;
        for ( i=0; i < n; i++ ) {
            j = PGAGetSortedPopIndex(ctx, i);
            PGACopyIndividual(ctx, j, PGA_OLDPOP, i, PGA_NEWPOP);
        }
        while (n < popsize) {
            m1 = PGASelectNextIndex(ctx, PGA_OLDPOP);
            m2 = PGASelectNextIndex(ctx, PGA_OLDPOP);
            if(PGARandomFlip(ctx, probcross)) {
                PGACrossover(ctx, m1, m2, PGA_OLDPOP, PGA_TEMP1, PGA_TEMP2, PGA_NEWPOP);
                PGAMutate (ctx,PGA_TEMP1,PGA_NEWPOP);
                PGAMutate (ctx,PGA_TEMP2,PGA_NEWPOP);
                PGACopyIndividual (ctx,PGA_TEMP1,PGA_NEWPOP,n, PGA_NEWPOP);
                PGACopyIndividual (ctx,PGA_TEMP2,PGA_NEWPOP,n+1,PGA_NEWPOP);
                n += 2;
            }
            else {
                PGACopyIndividual (ctx, m1, PGA_OLDPOP, n, PGA_NEWPOP);
                PGACopyIndividual (ctx, m2, PGA_OLDPOP, n+1, PGA_NEWPOP);
                n += 2;
            }
        }
        PGAEvaluate(ctx, PGA_NEWPOP, evaluate, NULL);
        PGAFitness (ctx, PGA_NEWPOP);
        PGAPrintReport(ctx, stdout, PGA_NEWPOP);
        PGAUpdateGeneration(ctx, NULL);
    }
    PGADestroy(ctx);
    return 0;
}

```

Figure 5.2: Example of Explicit Usage

5.4 Explicit PGAPack Functions

This section briefly discusses other functions not shown in the previous examples or discussed in Chapter 4. Additional information about these and other PGAPack functions is contained in Appendix B (function bindings) and the `./examples` directory.

`PGARunMutationAndCrossover` and `PGARunMutationOrCrossover` perform the recombination step. The former applies mutation to strings that undergo crossover. The latter applies only mutation to strings that did not undergo crossover. Note that this means that when `PGARunMutationAndCrossover` is selected, strings that are not crossed over (because the random process did not select the individuals for crossover with the given crossover probability) will also *not* be mutated! If no crossover is wanted, `PGARunMutationOnly` can be used for mutation only without crossover.

The restart operator described earlier in Section 4.8 can be invoked explicitly with `PGARestart(ctx, oldpop, newpop)`, where the best string from population `oldpop` is used to initialize population `newpop`.

`PGADuplicate(ctx, p, PGA_OLDDPOP, PGA_NEWPOP, 20)` returns `PGA_TRUE` if string `p` in population `PGA_OLDDPOP` is a duplicate of any of the first 20 strings in population `PGA_NEWPOP`. `PGACHange(ctx, p, PGA_OLDDPOP)` repeatedly applies the mutation operator to string `p` in population `PGA_OLDDPOP` until at least one mutation has occurred.

In PGAPack three values are associated with each string: (1) the user's evaluation function value, (2) a Boolean flag to indicate whether the evaluation function value is up to date with respect to the actual string, and (3) the fitness value. If `PGARun` is not used, the user must manage these values explicitly.

`PGAEvaluate(ctx, PGA_NEWPOP, evaluate, comm)` will execute the user's evaluation function, `evaluate`, on each string in population `PGA_NEWPOP` that has changed (for example, from crossover) since its last evaluation. `PGAEvaluate` will set both the evaluation function value and associated Boolean flag automatically. The argument `comm` is an MPI communicator. Valid values are `NULL` for an explicitly sequential example, or any valid MPI communicator. Depending on the number of processes specified when the program was invoked, and the value of the `comm` argument, `PGAEvaluate` may be run with one or more processes. See Chapter 9 for further discussion.

`PGAFitness` will calculate the population fitness values from the evaluation function values. It is an error to call `PGAFitness` if *all* the evaluation function values are not up to date.

These same three values may be read also. `PGAGetEvaluation(ctx, p, PGA_OLDDPOP)` returns the evaluation function value. `PGAGetEvaluationUpToDateFlag(ctx, p, PGA_OLDDPOP)` returns `PGA_TRUE` or `PGA_FALSE` to indicate whether the evaluation is up to date with the actual string or not, respectively. If PGAPack was compiled for debugging, `PGAGetEvaluation` will print a warning message if the evaluation is not up to date. `PGAGetFitness(ctx, p, PGA_OLDDPOP)` returns the fitness value.

At times, (e.g., applying a hill-climbing function) the user may need to explicitly set the evaluation function value and associated Boolean flag (fitness values can be calculated *only* by calling `PGAFitness`). `PGASetEvaluation(ctx, p, PGA_OLDDPOP, 123.4)` will set the evaluation function value to 123.4 and the associated Boolean flag to `PGA_TRUE`. The Boolean flag may be set independently with `PGASetEvaluationUpToDateFlag`. For example, `PGASetEvaluationUpToDateFlag(ctx, p, PGA_OLDDPOP, PGA_FALSE)` sets the status of the Boolean flag of string `p` in population `PGA_OLDDPOP` to out of date.

`PGAMean(ctx, a, n)` returns the mean of the `n` values in array `a`. `PGAStddev(ctx, a, n, mean)` returns the standard deviation of the `n` values in array `a` whose mean is `mean`. `PGARank(ctx, p, order, n)` returns an index that is the rank of string `p` as given by the sorted array `order` of length `n`.

`PGAGetPrintFrequency(ctx)` returns the frequency with which GA statistics are reported. `PGAGetWorstIndex(ctx, PGA_OLDDPOP)` returns the index of the string in population `PGA_OLDDPOP` with the worst evaluation function value. `PGAGetBestIndex(ctx, PGA_OLDDPOP)` returns the index of the string in population `PGA_OLDDPOP` with the best evaluation function value.

Chapter 6

Custom Usage: Native Data Types

This chapter discusses how PGAPack may be extended by replacing some of the standard PGAPack functions with user-defined functions for use with one of PGAPack's four *native* data types. This can be done from both C and Fortran.

6.1 Basics

In PGAPack, high-level (data-structure-neutral) functions call data-structure-specific functions that correspond to the data type used. The implementation uses function pointers that, by default, are set to the correct values for the datatype used. The user may change these defaults and set the function pointers to execute their functions instead. The functions the user can substitute for are initialization, crossover, mutation, checking for duplicate strings, string printing, termination criteria, a generic function called at the end of each GA iteration, and another generic function called *before* evaluation but after mutation and crossover.

The function call `PGASetUserFunction(ctx, PGA_USERFUNCTION_MUTATION, mymute)` will cause PGAPack to execute the function `mymute` whenever the mutation operator is called. Table 6.1 is a list of functions that can be customized for use with a native datatype. The first column describes the functionality, and the second column the symbolic constant for use with `PGASetUserFunction`. The calling sequence for these functions is fixed and *must* follow the function prototypes in Table 6.2. The files `./examples/templates/uf_native.c` and `./examples/templates/uf_native.f` contain template routines for these functions. A specific example is given below.

Checking the termination criteria requires some discussion. The function `PGADone` will *either* check to see if the standard stopping criteria (see Section 4.3) have been met, or call the user function specified by `PGA_USERFUNCTION_STOPCOND`. If you wish to have the user function check for the standard stopping criteria in addition to whatever else it does, it should call `PGACheckStoppingConditions(ctx)`. Do *not* call `PGADone` as this will cause an infinite loop to occur. Note that in a parallel program `PGACheckStoppingConditions`

Table 6.1: Customizeable Functions: Native Data Types

Functionality	Symbolic Constant
Initialization	PGA_USERFUNCTION_INITSTRING
Crossover	PGA_USERFUNCTION_CROSSOVER
Mutation	PGA_USERFUNCTION_MUTATION
Duplicate Checking	PGA_USERFUNCTION_DUPLICATE
String Printing	PGA_USERFUNCTION_PRINTSTRING
Termination Criteria	PGA_USERFUNCTION_STOPCOND
End of generation	PGA_USERFUNCTION_ENDOFGEN
Genetic difference	PGA_USERFUNCTION_GEN_DIFFERENCE
Pre-Evaluate Hook	PGA_USERFUNCTION_PRE_EVAL

Table 6.2: Calling Sequences for Customizable Functions

Symbolic Constant	Return	Function Prototype
PGA_USERFUNCTION_INITSTRING	void	(PGAContext*, int, int)
PGA_USERFUNCTION_CROSSOVER	void	(PGAContext*, int, int, int, int, int, int)
PGA_USERFUNCTION_MUTATION	int	(PGAContext*, int, int, double)
PGA_USERFUNCTION_DUPLICATE	int	(PGAContext*, int, int, int, int)
PGA_USERFUNCTION_PRINTSTRING	void	(PGAContext*, FILE *, int, int)
PGA_USERFUNCTION_STOPCOND	int	(PGAContext*)
PGA_USERFUNCTION_ENDOFGEN	void	(PGAContext*)
PGA_USERFUNCTION_GEN_DIFFERENCE	double	(PGAContext*, int, int, int, int)
PGA_USERFUNCTION_PRE_EVAL	void	(PGAContext*, int)

should only be called by the master process (see Chapter 9).

The end of generation function (which is null by default) may be used for gathering statistics about the GA, displaying custom output, etc. This function is called after all generational computation is complete, but before the population pointers (PGA_NEWPOP, PGA_OLDDPOP) have been switched and the standard PGAPack output printed. Therefore, be sure to use PGA_NEWPOP as the population pointer. There is no mechanism for suppressing the standard PGAPack generational output.

The genetic difference function computes the genetic difference of two individuals. It is only used when restricted tournament selection is in use. There are implementations for the standard data types: For binary alleles it uses the hamming distance. For real- and integer valued genes it uses an allele-by-allele absolute value of the difference by default (also know as Manhattan distance), i.e.

$$\sum_{i=0}^{s-1} |a_{ij} - a_{ik}|$$

where s is the string length, a_{ij} and a_{ik} are the alleles to be compared. For character alleles it counts the number of differences. You can set the difference function for integer and real data types to an Euclidian distance by calling, e.g., for the real data type:

```
PGASetUserFunction(ctx, PGARealEuclidianDistance);
```

or for the integer data type:

```
PGASetUserFunction(ctx, PGAIntegerEuclidianDistance);
```

This will use the Euclidian distance:

$$\sqrt{\sum_{i=0}^{s-1} (a_{ij} - a_{ik})^2}$$

When using user-defined data types together with restricted tournament selection, an implementation of the distance function for the user-defined data type has to be provided.

The Pre-Evaluate Hook function can be used for performing actions that need to be done before evaluating the generated individuals after crossover and mutation. It can be used, e.g., for repairing genes after crossover and mutation before evaluating them. It can also be used in concert with the end of generation function to perform caching of evaluations: The end of generation function would cache genes and their evaluation while the pre evaluate hook would look up newly-generated individuals in the cache: If a newly-generated individual is found in the cache, an evaluation can be saved which may have an impact on the runtime if evaluation is costly. Note that the probability of cache hits may be higher for binary and integer alleles than for real alleles. Note that for non-parallel implementations, caching could also be implemented in the evaluate function but for parallel implementations this would not work because each parallel instance would use a separate cache. The pre evaluation user function is called only in the master instance for a parallel implementation.

6.2 Example Problem: C

The example problem in Figure 6.1 is to maximize $\sum_{j=1}^L x_j$ with $1 \leq x_j \leq L$, where L is the string length. The optimal solution to this problem, L^2 , is achieved by setting each x_j to L . The files for this example, `./examples/maxint.c` and `./examples/maxint.f`, contain template routines for these functions.

The example shows the use of a custom mutation function with an integer data type. The function `PGASetUserFunction` specifies that this function, `MyMutation`, will be called when the mutation operator is applied, rather than the default mutation operator. `MyMutation` generates a random integer on the interval $[1, L]$.

6.3 Example Problem: Fortran

Figure 6.2 is the same example as in Figure 6.1 written in Fortran.

```

#include <pgapack.h>

double evaluate (PGAContext *ctx, int p, int pop, double *aux);
int myMutation (PGAContext *ctx, int p, int pop, double pm);

int main( int argc, char **argv )
{
    PGAContext *ctx;
    int i, maxiter;
    ctx = PGACreate (&argc, argv, PGA_DATATYPE_INTEGER, 10, PGA_MAXIMIZE);
    PGASetUserFunction (ctx, PGA_USERFUNCTION_MUTATION, myMutation);
    PGASetIntegerInitPermute(ctx, 1, 10);
    PGASetUp (ctx);
    PGARun (ctx, evaluate);
    PGADestroy (ctx);
    return(0);
}

int myMutation(PGAContext *ctx, int p, int pop, double pm)
{
    int stringlen, i, k, count = 0;
    stringlen = PGAGetStringLength(ctx);
    for (i = 0; i < stringlen; i++)
        if (PGARandomFlip(ctx, pm)) {
            k = PGARandomInterval(ctx, 1, stringlen);
            PGASetIntegerAllele(ctx, p, pop, i, k);
            count++;
        }
    return ((double) count);
}

double evaluate(PGAContext *ctx, int p, int pop, double *aux)
{
    int stringlen, i, sum = 0;
    stringlen = PGAGetStringLength(ctx);
    for (i = 0; i < stringlen; i++)
        sum += PGAGetIntegerAllele(ctx, p, pop, i);
    return ((double)sum);
}

```

Figure 6.1: PGAPack C Example Using Custom Mutation Operator

```

include 'pgapackf.h'
include 'mpif.h'
double precision evaluate
integer          myMutation
external         evaluate, myMutation
integer ctx, i, maxiter, ierror
call MPI_Init(ierr)
ctx = PGACreate (PGA_DATATYPE_INTEGER, 10, PGA_MAXIMIZE)
call PGASetUserFunction      (ctx, PGA_USERFUNCTION_MUTATION,
&    myMutation)
call PGASetIntegerInitPermute(ctx, 1, 10);
call PGASetUp                (ctx);
call PGARun                  (ctx, evaluate);
call PGADestroy              (ctx);
call MPI_Finalize(ierr)
stop
end

integer function myMutation(ctx, p, pop, pm)
include      'pgapackf.h'
integer      ctx, p, pop
double precision  pm
integer      stringlen, i, k, count
count = 0
stringlen = PGAGetStringLength(ctx)
do i=0, stringlen
  if (PGARandomFlip(ctx, pm) .eq. PGA_TRUE) then
    k = PGARandomInterval(ctx, 1, stringlen)
    call PGASetIntegerAllele(ctx, p, pop, i, k)
    count = count + 1
  endif
enddo
myMutation = count
return
end

double precision function evaluate(ctx, p, pop)
include      'pgapackf.h'
integer ctx, p, pop
integer      stringlen, i, sum
sum = 0
stringlen = PGAGetStringLength(ctx)
do i=0, stringlen
  sum = sum + PGAGetIntegerAllele(ctx, p, pop, i)
enddo
evaluate = sum
return
end

```

Figure 6.2: PGAPack Fortran Example Using Custom Mutation Operator

Chapter 7

Custom Usage: New Data Types

This chapter discusses how PGAPack may be extended by defining a new data type. Defining a new data type may be done only in C programs.

7.1 Basics

To create a new data type, you must (1) specify `PGA_DATATYPE_USER` for the datatype in the `PGACreate` call and (2) for *each* entry in Table 7.1, call `PGASetUserFunction` to specify the function that will perform the given operation on the new data type. If the data type is `PGA_DATATYPE_USER`, the string length specified to `PGACreate` can be whatever the user desires. It will be returned by `PGAGetStringLength` but is not otherwise used in the data-structure-neutral functions of PGAPack.

The calling sequences for the functions in Table 7.1 are given in Table 7.2. Template routines for these functions are in the file `./examples/templates/uf_new.c`.

While PGAPack requires that the user supply all the functions in Table 7.1, your program may not require the functionality of all of them. For example, the user really does not need to write a function to pack the strings for message-passing unless a parallel version of PGAPack is being used. In these cases, we suggest that the user supply a stub function; i.e., a function with the correct calling sequence but no functionality.

7.2 Example Problem

This example illustrates use of a user-defined structure as the new data type. The problem is one of molecular docking where one protein molecule (the ligand) is to be docked into a second, target protein molecule. Figure 7.1 contains the function prototypes for each function that will operate on the new datatype, the definition of the user's structure (`ligand`), and the main program.

The first three `doubles` of the array `t` in structure `ligand` represent the translation of the ligand molecule in the *x*-, *y*-, and *z*-axes, respectively. The last three `doubles` in the array `t` represent the rotation of the

Table 7.1: Functions Required for New Data Types

Functionality	Symbolic Constant
Memory allocation	<code>PGA_USERFUNCTION_CREATESTRING</code>
String packing	<code>PGA_USERFUNCTION_BUILDDATATYPE</code>
Mutation	<code>PGA_USERFUNCTION_MUTATION</code>
Crossover	<code>PGA_USERFUNCTION_CROSSOVER</code>
String printing	<code>PGA_USERFUNCTION_PRINTSTRING</code>
String copying	<code>PGA_USERFUNCTION_COPYSTRING</code>
Duplicate checking	<code>PGA_USERFUNCTION_DUPLICATE</code>

Table 7.2: Calling Sequences for New Data Type Functions

Symbolic Constant	Return	Function Prototype
PGA_USERFUNCTION_CREATESTRING	void	(PGAContext*, int, int, int)
PGA_USERFUNCTION_BUILDDATATYPE	int	(PGAContext*, int, int)
PGA_USERFUNCTION_MUTATION	int	(PGAContext*, int, int, double)
PGA_USERFUNCTION_CROSSOVER	void	(PGAContext*, int, int, int, int, int, int)
PGA_USERFUNCTION_PRINTSTRING	void	(PGAContext*, FILE *, int, int)
PGA_USERFUNCTION_COPYSTRING	int	(PGAContext*, int, int, int, int)
PGA_USERFUNCTION_DUPLICATE	int	(PGAContext*, int, int, int, int)

```

#include <pgapack.h>

double      energy      (double *, int *);
double      Evaluate    (PGAContext *, int, int);
void        CreateString (PGAContext *, int, int, int);
int         Mutation     (PGAContext *, int, int, double);
void        Crossover    (PGAContext *, int, int, int, int, int, int);
void        WriteString  (PGAContext *, FILE *, int, int);
void        CopyString   (PGAContext *, int, int, int, int);
int         DuplicateString (PGAContext *, int, int, int, int);
MPI_Datatype BuildDT     (PGAContext *, int, int);

typedef struct {
    double t[6];          /* ligand translation and rotation */
    int    sc[40];        /* ligand sidechain rotations      */
} ligand;

int main(int argc, char **argv) {
    PGAContext *ctx;
    int maxiter;
    ctx = PGACreate(&argc, argv, PGA_DATATYPE_USER, 46, PGA_MINIMIZE);
    PGASetRandomSeed (ctx, 1);
    PGASetMaxGAIterValue(ctx, 5000);
    PGASetUserFunction (ctx, PGA_USERFUNCTION_CREATESTRING, CreateString);
    PGASetUserFunction (ctx, PGA_USERFUNCTION_MUTATION, Mutation);
    PGASetUserFunction (ctx, PGA_USERFUNCTION_CROSSOVER, Crossover);
    PGASetUserFunction (ctx, PGA_USERFUNCTION_PRINTSTRING, WriteString);
    PGASetUserFunction (ctx, PGA_USERFUNCTION_COPYSTRING, CopyString);
    PGASetUserFunction (ctx, PGA_USERFUNCTION_DUPLICATE, DuplicateString);
    PGASetUserFunction (ctx, PGA_USERFUNCTION_BUILDDATATYPE, BuildDT);
    PGASetUp (ctx);
    PGARun (ctx, Evaluate);
    PGADestroy (ctx);
    return (0);
}

```

Figure 7.1: Main Program for Structure Data Type

```

void CreateString(PGAContext *ctx, int p, int pop, int InitFlag) {
    int i;
    ligand *ligand_ptr;
    PGAIndividual *new;

    new = PGAGetIndividual(ctx, p, pop);
    if (!(new->chrom = malloc(sizeof(ligand)))) {
        fprintf(stderr, "No room for new->chrom");
        exit(1);
    }
    ligand_ptr = (ligand *)new->chrom;
    if (InitFlag) {
        for (i = 0; i < 3; i++)
            ligand_ptr->t[i] = PGARandom01(ctx, 0) * 20.0 - 10.0;
        for (i = 3; i < 6; i++)
            ligand_ptr->t[i] = PGARandom01(ctx, 0) * 6.28 - 3.14;
        for (i = 0; i < 40; i++)
            ligand_ptr->sc[i] = PGARandomInterval(ctx, -20, 20);
    } else {
        for (i = 0; i < 6; i++)
            ligand_ptr->t[i] = 0.0;
        for (i = 0; i < 40; i++)
            ligand_ptr->sc[i] = 0;
    }
}

```

Figure 7.2: Creation and Initialization Function for Structure Data Type

ligand molecule about each of the axes. The `ints` in the `sc` array represent side chain rotations (which are discrete) of the ligand molecule.

Figure 7.2 contains the function `CreateString` that allocates and initializes the ligand structure. At this level of usage it is no longer always possible to maintain the `(p,pop)` abstraction to specify an individual (a string and associated fields). `CreateString` works directly with the string pointer that `(p,pop)` is mapped to. If `InitFlag` is true, `CreateString` will initialize the fields; otherwise they are set to 0.

`PGAGetIndividual(ctx, p, pop)` returns a pointer of type `PGAIndividual` to the individual (the string and associated fields) specified by `(p,pop)`. `PGAIndividual` is a structure, one of the fields of which is `chrom`, a void pointer to the string itself. That pointer, `new->chrom`, is assigned the address of the memory allocated by the `malloc` function. As `malloc` returns a void pointer, no cast is necessary.

The value of `InitFlag` is passed by `PGAPack` to the user's string creation routine. It specifies whether to randomly initialize the string or set it to zero. By default, `PGA_OLDDPOP` (except for `PGA_TEMP1` and `PGA_TEMP1` which are set to zero) is randomly initialized, and `PGA_NEWPOP` is set to zero. This choice may be changed with the `PGASetRandomInitFlag` function discussed in Section 4.4.)

Figure 7.3 contains the mutation function `Mutation` for the ligand data type. Each of the 46 genes has a probability of `mr` of being changed. If a mutation occurs, `Mutation` adds or subtracts one tenth to the existing value of a `double`, and adds or subtracts one to an `int`.

Figure 7.4 contains the crossover function `Crossover`, which implements uniform crossover for the ligand data type. The lines

```

parent1 = (ligand *)PGAGetIndividual(ctx, p1, pop1)->chrom;
parent2 = (ligand *)PGAGetIndividual(ctx, p2, pop1)->chrom;
child1  = (ligand *)PGAGetIndividual(ctx, t1, pop2)->chrom;
child2  = (ligand *)PGAGetIndividual(ctx, t2, pop2)->chrom;

```

are worthy of mention. Each implements in one line what the two lines

```

int Mutation(PGAContext *ctx, int p, int pop, double mr) {
    ligand *ligand_ptr;
    int i, count = 0;

    ligand_ptr = (ligand *)PGAGetIndividual(ctx, p, pop)->chrom;
    for (i = 0; i < 6; i++)
        if (PGARandomFlip(ctx, mr)) {
            if (PGARandomFlip(ctx, 0.5))
                ligand_ptr->t[i] += 0.1*ligand_ptr->t[i];
            else
                ligand_ptr->t[i] -= 0.1*ligand_ptr->t[i];
            count++;
        }
    for (i = 0; i < 40; i++)
        if (PGARandomFlip(ctx, mr)) {
            if (PGARandomFlip(ctx, 0.5))
                ligand_ptr->sc[i] += 1;
            else
                ligand_ptr->sc[i] -= 1;
            count++;
        }
    return (count);
}

```

Figure 7.3: Mutation for Structure Data Type

```

new = PGAGetIndividual(ctx, p, pop);
string = (ligand *)new->chrom;

```

in `Mutation` did. Either style is acceptable. `PGAGetIndividual` returns a pointer whose `chrom` field (a `void` pointer) is cast to the `ligand` data type.

Figure 7.5 contains the code for `DuplicateString`, which checks for duplicate ligand structures. It uses the ANSI C `memcmp` function for this purpose.

Figure 7.6 contains the evaluation function for this example. It again uses `PGAGetIndividual` to map `(p, pop)` into a pointer to the string of interest. For user data types, `PGAPack` does *not* provide a `PGAGetUserAllele` function, so access to the allele values is made directly through the pointer.

Figure 7.7 contains the function `BuildDT` that builds an MPI datatype for sending strings to other processors. Consult an MPI manual for further information.

```

void Crossover(PGAContext *ctx, int p1, int p2, int pop1, int t1, int t2,
              int pop2) {
    int i;
    ligand *parent1, *parent2, *child1, *child2;
    double pu;

    parent1 = (ligand *)PGAGetIndividual(ctx, p1, pop1)->chrom;
    parent2 = (ligand *)PGAGetIndividual(ctx, p2, pop1)->chrom;
    child1  = (ligand *)PGAGetIndividual(ctx, t1, pop2)->chrom;
    child2  = (ligand *)PGAGetIndividual(ctx, t2, pop2)->chrom;

    pu = PGAGetUniformCrossoverProb(ctx);

    for (i = 0; i < 6; i++)
        if (PGARandomFlip(ctx, pu)) {
            child1->t[i] = parent1->t[i];
            child2->t[i] = parent2->t[i];
        } else {
            child1->t[i] = parent2->t[i];
            child2->t[i] = parent1->t[i];
        }
    for (i = 0; i < 40; i++)
        if (PGARandomFlip(ctx, pu)) {
            child1->sc[i] = parent1->sc[i];
            child2->sc[i] = parent2->sc[i];
        } else {
            child1->sc[i] = parent2->sc[i];
            child2->sc[i] = parent1->sc[i];
        }
}

```

Figure 7.4: Crossover for Structure Data Type

```

int DuplicateString(PGAContext *ctx, int p1, int pop1, int p2, int pop2) {
    void *a, *b;

    a = PGAGetIndividual(ctx, p1, pop1)->chrom;
    b = PGAGetIndividual(ctx, p2, pop2)->chrom;
    return (!memcmp(a, b, sizeof(ligand)));
}

```

Figure 7.5: Duplicate Testing for Structure Data Type

```

double Evaluate(PGAContext *ctx, int p, int pop, double *aux) {
    int i, j;
    double x[6];
    int sc[40];
    PGAIndividual *ind;
    ligand *lig;

    lig = (ligand *)PGAGetIndividual(ctx, p, pop)->chrom;
    for (i = 0; i < 6; i++)
        x[i] = lig->t[i];
    for (i = 0; i < 40; i++)
        sc[i] = lig->sc[i];
    return ( energy(x,sc) );
}

```

Figure 7.6: Evaluation Function for Structure Data Type

```

MPI_Datatype BuildDT(PGAContext *ctx, int p, int pop) {
    int            counts[5];
    MPI_Aint       displs[5];
    MPI_Datatype   types[5];
    MPI_Datatype   DT_PGAIindividual;
    PGAIndividual  *P;
    ligand         *S;

    P = PGAGetIndividual(ctx, p, pop);
    S = (ligand *)P->chrom;

    /* Build the MPI datatype. Every user defined function needs these.
     * The first two calls are stuff that is internal to PGAPack, but
     * the user still must include it. See pgapack.h for details one the
     * fields (under PGAIndividual)
     */
    MPI_Address(&P->evalfunc, &displs[0]);
    counts[0] = 2;
    types[0] = MPI_DOUBLE;

    /* Next, we have an integer, evaluptodate. */
    MPI_Address(&P->evaluptodate, &displs[1]);
    counts[1] = 1;
    types[1] = MPI_INT;

    /* Finally, we have the actual user-defined string. */
    MPI_Address(S->t, &displs[2]);
    counts[2] = 6;
    types[2] = MPI_DOUBLE;

    MPI_Address(S->sc, &displs[3]);
    counts[3] = 40;
    types[3] = MPI_INT;

    MPI_Type_struct(4, counts, displs, types, &DT_PGAIindividual);
    MPI_Type_commit(&DT_PGAIindividual);
    return(DT_PGAIindividual);
}

```

Figure 7.7: Message Packing Function for Structure Data Type

Chapter 8

Hill-Climbing and Hybridization

Hill-climbing heuristics attempt to improve a solution by moving to a better *neighbor* solution. Whenever the neighboring solution is better than the current solution, it replaces the current solution. Genetic algorithms and hill-climbing heuristics have complementary strong and weak points. GAs are good at finding promising areas of the search space, but not as good at fine-tuning within those areas. Hill-climbing heuristics, on the other hand, are good at fine-tuning, but lack a global perspective. Practice has shown that a *hybrid* algorithm that combines GAs with hill-climbing heuristics often results in an algorithm that can outperform either one individually.

There are two general schemes for creating hybrid algorithms. The simplest is to run the genetic algorithm until it terminates and then apply a hill-climbing heuristic to each (or just the best) string. The second approach is to integrate a hill-climbing heuristic with the genetic algorithm. Choices to be made in the second case include how often to apply the hill-climbing heuristic and how many strings in the population to apply it to.

PGAPack supports hybrid schemes in the following ways:

- By passing the context variable as a parameter to the user's hill-climbing function, the user has access to solution and parameter values, debug flags, and other information.
- The functions `PGAGetBinaryAllele`, `PGAGetIntegerAllele`, `PGAGetRealAllele`, and `PGAGetCharacterAllele` allow the user's hill-climbing function to read allele values, and the functions `PGASetBinaryAllele`, `PGASetIntegerAllele`, `PGASetRealAllele`, and `PGASetCharacterAllele` allow the user's hill-climbing function to set allele values explicitly.
- The functions `PGADecodeRealAsBinary`, `PGADecodeRealAsGrayCode`, `PGADecodeIntegerAsBinary`, and `PGADecodeIntegerAsGrayCode` allow the user's hill-climbing function to read integer or real numbers encoded as binary or Gray code strings.
- The functions `PGAEncodeRealAsBinary`, `PGAEncodeRealAsGrayCode`, `PGAEncodeIntegerAsBinary`, and `PGAEncodeIntegerAsGrayCode` allow the user's hill-climbing function to encode integer or real numbers as binary or Gray code strings.
- The functions `PGAGetEvaluation` and `PGASetEvaluation` allow the user's hill-climbing function to get and set evaluation function values, and `PGASetEvaluationUpToDateFlag` and `PGAGetEvaluationUpToDateFlag` to get and set the flag that indicates whether an evaluation function value is up to date. These functions have an optional fifth argument `aux`, that allows to get or set the auxiliary evaluations, see section 4.9.

One way to run a hybrid GA and use `PGARun` is to use the `PGASetUserFunction` discussed in Chapter 6 to specify a user function to be called at the end of each GA iteration. A more flexible approach would be for the user to call the high-level PGAPack functions, and their hillclimber to explicitly specify the steps of the hybrid GA.

Figure 8.1 is a version of the Maxbit problem given in Section 2.1. It uses the hill-climbing function `hillclimb`, which is called after the recombination step. It randomly selects a gene to set to one. Note the

PGASetEvaluationUpToDateFlag call. It sets the flag that indicates the evaluation function is not current with the string (since the string was changed). It is *critical* that this flag be set when the user changes a string, since the value of this flag determines whether PGAEvaluate will invoke the user's function evaluation routine.

```
#include "pgapack.h"

double evaluate(PGAContext *, int, int, double *aux);
void hillclimb (PGAContext *, int);

int main(int argc, char **argv)
{
    PGAContext *ctx;

    ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
    PGASetUp (ctx);
    PGAEvaluate(ctx, PGA_OLDPOP, evaluate, NULL);
    PGAFitness (ctx, PGA_OLDPOP);
    while(!PGADone(ctx, NULL)) {
        PGASelect (ctx, PGA_OLDPOP);
        PGARunMutationAndCrossover(ctx, PGA_OLDPOP, PGA_NEWPOP);
        hillclimb (ctx, PGA_NEWPOP);
        PGAEvaluate (ctx, PGA_NEWPOP, evaluate, NULL);
        PGAFitness (ctx, PGA_NEWPOP);
        PGAUpdateGeneration (ctx, NULL);
        PGAPrintReport (ctx, stdout, PGA_OLDPOP);
    }
    PGADestroy(ctx);
    return 0;
}

void hillclimb(PGAContext *ctx, int pop)
{
    int i, p, popsize, stringlen;
    popsize = PGAGetPopSize(ctx);
    stringlen = PGAGetStringLength(ctx);
    for (p=0; p<popsize; p++) {
        i = PGARandomInterval(ctx, 0, stringlen-1);
        PGASetBinaryAllele (ctx, p, pop, i, 1);
        PGASetEvaluationUpToDateFlag (ctx, p, pop, PGA_FALSE);
    }
}
```

Figure 8.1: Hill-Climbing Heuristic for Maxbit Example

Chapter 9

Parallel Aspects

This chapter assumes familiarity with the background material in Appendix C. It also assumes that a parallel version of PGAPack was built and that programs are linked with an MPI library.

Version 1.0 of PGAPack supports parallel and sequential *implementations* of the single population global model (GM). The parallel implementation uses a master/slave algorithm in which one process, the *master*, executes all steps of the genetic algorithm *except* the function evaluations. The function evaluations are executed by the *slave* processes¹.

9.1 Basic Usage

Both sequential and parallel versions of PGAPack may be run by using `PGARun`. The choice of sequential or parallel execution depends on the number of processes specified when the program is started. If one process is specified, the sequential implementation of the GM is used (even in a parallel version of PGAPack). If two or more processes are specified, the parallel implementation of the GM is used. The examples in Chapter 2 can all be run in parallel by specifying more than one process at startup.

The specification of the number of processors is done at run time. The actual format of the specification depends on the MPI implementation and computer used (see Appendix C for some examples). `PGARun` uses the default MPI communicator, `MPI_COMM_WORLD`. This specifies that all processes specified at startup participate in the computation: one as the master process, the others as slave processes. A different communicator may be specified with `PGASetCommunicator(ctx, comm)`, where `comm` is an MPI communicator.

`PGARun` is really a “wrapper” function that calls `PGARunGM` with the `MPI_COMM_WORLD` communicator. The user may call `PGARunGM` directly, that is, `PGARunGM(ctx, evaluate, MPI_COMM_WORLD)` where `evaluate` is the name of the user’s evaluation function and the third argument is an MPI communicator. Note that the communicator specified by `PGASetCommunicator` does *not* affect `PGARunGM`.

9.2 Explicit Use

In general, explicit use of the parallel features is more complicated than in the case of sequential functions. This is because the user’s program must coordinate the execution threads of *multiple* processes. `PGARunGM` encapsulates all that is necessary into one routine, and parts of its source code may serve as a useful starting point if one wishes to develop an explicitly parallel program. The parallel functions in PGAPack may be viewed as a hierarchy with `PGARun` and `PGARunGM` at the top of the hierarchy, `PGAEvaluate` next, `PGASendIndividual`, `PGARecieveIndividual`, and `PGASendReceiveIndividual` next, and `PGABuildDatatype` at the bottom of the hierarchy.

`PGAGetRank(ctx, comm)` returns the rank of the process in communicator `comm`. If `comm` is `NULL` it returns 0. `PGAGetNumProcs(ctx, comm)` returns the number of processes in communicator `comm`. If `comm` is `NULL` it returns 1.

¹In the special case of exactly two processes, the master executes function evaluations as well.

The type of algorithm used to execute `PGAEvaluate(ctx,pop,f,comm)` will depend on the number of processes in the communicator `comm`. If it is `NULL` or contains one process, a sequential implementation will be used. If more than one process is specified it will execute a master/slave evaluation of the strings in population `pop` that require evaluation by applying, `f`, the user's evaluation function. `PGAEvaluate` should be called by *all* processes in communicator `comm`.

`PGASendIndividual(ctx,p,pop,dest,tag,comm)` will send string `p` in population `pop` to process `dest`. `tag` is a tag used to identify the message, and `comm` is an MPI communicator. This function calls `MPI_Send` to perform the actual message passing. In addition to string `p` itself, the evaluation function value, fitness function value, and evaluation status flag are also sent.

`PGAResceiveIndividual` is the complementary function to `PGASendIndividual`. For example, `PGAResceiveIndividual(ctx,p,pop,source,tag,comm,status)` will store in location `p` in population `pop` the string and fields of the individual sent from process `source` with the MPI tag `tag` and MPI communicator `comm`. `status` is an MPI status vector.

`PGASendReceiveIndividual` combines the functionality of `PGASendIndividual` and `PGAResceiveIndividual`. This may be useful in avoiding potential deadlock on some systems. For example, `PGASendReceiveIndividual(ctx,sp,spop,dest,stag,rp,rpop,source,rtag,comm,status)`. Here, `sp` is the index of the string in population `spop` to send to process `dest` with tag `stag`. The string received from process `source` with tag `rtag` is stored in location `rp` in population `rpop`. `comm` and `status` are the same as defined earlier.

`PGABuildDatatype(ctx,p,pop)` packs together the string and fields that `PGASendIndividual`, `PGAResceiveIndividual`, and `PGASendReceiveIndividual` send and receive. The result is of type `MPI_Datatype`.

9.3 Example

Figure 9.1 is a parallel version of the example in Figure 5.1. Since we now have *multiple* processes executing the program at the same time, we must coordinate each ones execution. In the example, the master process (the one with rank 0 as determined by `PGAGetRank`) executes all functions, and the slave processes execute only those functions that take a communicator as an argument. Note that this example will execute correctly even if only one process is in the communicator.

9.4 Performance

The parallel implementation of the GM will produce the *same* result as the sequential implementation, usually faster. However, the parallel implementation varies with the number of processes. 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 is responsible for bookkeeping only, and the slaves for executing the function evaluations. In general, the speedup obtained will vary with the amount of computation associated with a function evaluation and the computational overhead of distributing and collecting information to and from the slave processes.

The speedup that can be achieved with the master/slave model is limited by the number of function evaluations that can be executed in parallel. This number depends on the population size and the number of new strings created each generation. For example, if the population size is 100 and a 100 new strings are created each GA generation, then up to 100 processors can be put to effective use to run the slave processes. However, with the default rule of replacing only 10% of the population each GA generation, only 10 processors can be used effectively.

```

#include "pgapack.h"
double evaluate (PGAContext *ctx, int p, int pop, double *aux);

int main(int argc, char **argv)
{
    PGAContext *ctx;
    int rank;

    ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
    PGASetUp (ctx);
    rank = PGAGetRank(ctx, MPI_COMM_WORLD);
    PGAEvaluate(ctx, PGA_OLDPOP, evaluate, MPI_COMM_WORLD);
    if ( rank == 0 )
        PGAFitness (ctx, PGA_OLDPOP);
    while(!PGADone(ctx, MPI_COMM_WORLD)) {
        if ( rank == 0 ) {
            PGASelect (ctx, PGA_OLDPOP);
            PGARunMutationAndCrossover(ctx, PGA_OLDPOP, PGA_NEWPOP);
        }
        PGAEvaluate(ctx, PGA_OLDPOP, evaluate, MPI_COMM_WORLD);
        if ( rank == 0 )
            PGAFitness (ctx, PGA_NEWPOP);
        PGAUpdateGeneration (ctx, MPI_COMM_WORLD);
        if ( rank == 0 )
            PGAPrintReport (ctx, stdout, PGA_OLDPOP);
    }
    PGADestroy(ctx);
    return(0);
}

```

Figure 9.1: Simple Parallel Example of Explicit Usage

Chapter 10

Fortran Interface

PGAPack is written entirely in ANSI C. A set of interface functions, also written in C, is designed to be called by Fortran programs and then call the “real” C routine. This mechanism provides most of PGAPack’s functionality to Fortran programs. The following list contains most major differences between C and Fortran. Additional, machine-specific idiosyncrasies are noted in Appendix D.

- The `Makefiles` for the Fortran examples (in `./examples/fortran` and `./examples/mgh`) are *not* configured to use the `-I` mechanism for specifying the include file search path (since not all Fortran compilers support this). Therefore, you will need to copy or set up a symbolic link to `./include/pgapackf.h` from the directory you are compiling a Fortran program in.
- The context variable is declared `integer` (or `integer*8`, see Appendix D) in Fortran.
- `PGACreate` takes only three arguments in Fortran (not `argc` or `argv` as in C).
- The Fortran include file is `pgapackf.h` and should be included in any Fortran subroutine or function that calls a PGAPack function, to ensure correct typing and definition of functions and symbolic constants.
- If a C function returns an `{ int, double, pointer }`, the corresponding Fortran function returns an `{ integer, double precision, integer }`. If the C function is `void` it is implemented as a Fortran subroutine.
- When supplying function arguments, a C `int` corresponds to a Fortran `integer`, and a C `double` corresponds to a Fortran `double precision`. For example, to set the crossover probability to 0.6, use `call PGASetCrossoverProb(ctx, 0.6d0)`,
or

```
double precision pc
pc = 0.6
call PGASetCrossoverProb(ctx, pc)
```
- Gene indices are `[0, L - 1]` in C, and `[1, L]` in Fortran, where `L` is the string length.
- Population member indices are `[0, N - 1]` in C, and `[1, N]` in Fortran, where `N` is the population size.
- Fortran does not support command line arguments (Section 4.13).
- Fortran allows custom usage with native data types (Chapter 6), but not with new data types (Chapter 7).
- In the MPICH implementation of MPI, the Fortran and C versions of `MPI_Init` are different. If the main program is in C, then the C version of `MPI_Init` must be called. If the main program is in Fortran, the Fortran version of `MPI_Init` must be called. Therefore, Fortran users of PGAPack with MPICH must call `MPI_Init` themselves since `PGACreate`, which calls `MPI_Init` if users haven’t called it themselves, is written in C.

- The DEC Alpha and Silicon Graphics Power Challenge, which have 64-bit C pointers and 32-bit Fortran integers (but not the Cray T3D which has 64-bit Fortran integers), have additional differences¹. These arise because a Fortran integer is too small to hold the address returned by the C interface routine.

- The context variable should be declared `integer*8`.
- `MPI_COMM_WORLD` should *not* be passed directly to PGAPack Fortran functions. Instead, `PGAGetCommunicator` should be called to return the address into an `integer*8` variable. For example

```
integer pid
integer*8 comm
comm = PGAGetCommunicator(ctx)
:
pid = PGAGetRank(ctx, comm)
```

- `MPI_COMM_WORLD` *can* and *should* be passed directly to any MPI routines called directly from Fortran.
- Calling an MPI routine that returns a communicator is safe. However, passing the returned communicator to a PGAPack Fortran function will usually fail.

¹More generally, these issues arise whenever the size of a Fortran integer is less than the size of a pointer.

Chapter 11

Debugging Tools

PGAPack has a sophisticated built-in trace facility that is useful for debugging. When the facility is invoked, print statements to `stdout` allow the programmer to trace the sequence of functions PGAPack executes. Due to the negative impact on performance this facility is *not* available by default. Instead, you must explicitly enable tracing when configuring PGAPack with the `-debug` flag.

The trace facility uses the concept of a *debug level*. For example, executing the Maxbit example (Figure 2.1) with a debug level of 12, `maxbit -pgadb 12`, will print the output shown in Figure 11.1. The “0:” is the process rank. This is followed by the name of a PGAPack function and the “action” that caused the print statement to execute. In this case, the action is entering the function. Note that the rank printed for a process is *always* its rank in the `MPI_COMM_WORLD` communicator, even if another communicator was set.

Tracing is enabled by specifying one or more debug levels to trace. A list of debug levels is given in Table 11.1. Not all debug level values are currently used. The values 1–10 are reserved for users as described below.

C programmers may set the debug level from the command line using either `-pgadb <debug_level>` or `-pgadebug <debug_level>`. Several forms of the `<debug_level>` argument are allowed. `-pgadb 12` will trace entering all high-level functions as shown in Figure 11.1. `-pgadb 12,13` or `-pgadb 12-13` will trace entering *and* exiting of all high-level functions. The command line option `-pgahelp debug` will list the debug level options and then exit.

Fortran (and C) programmers may access the trace facility via function calls. The function `PGASetDebugLevel` may be called to set a debug level. For example, call `PGASetDebugLevel(ctx,12)` would produce the same output shown in Figure 11.1. `PGAClearDebugLevel(ctx,12)` will clear prints associated with debug level 12. `PGAPrintDebugOptions(ctx)` will print the list of available debug options.

The function `PGASetDebugLevelByName` will turn on debugging of the named function. For example, `PGASetDebugLevelByName(ctx, 'PGACrossover')` will enable all the trace prints of `PGACrossover`. `PGAClearDebugLevelByName` will disable the tracing of the specified function.

Users can use the trace facility in their own functions (e.g., their evaluation function) in two ways. First, they can insert `PGADebugPrint` function calls in their functions using one of the symbolic constants defined in the header file `pgapack.h`. These are `PGA_DEBUG_ENTERED`, `PGA_DEBUG_EXIT`, `PGA_DEBUG_MALLOC`, `PGA_DEBUG_PRINTVAR`, `PGA_DEBUG_SEND`, and `PGA_DEBUG_RECV` for entering a function, exiting a function, allocating memory, print a variable’s value, and sending or receiving a string, respectively.

For example, `PGADebugPrint(ctx, PGA_DEBUG_ENTERED, "MyFunc", "Entered", PGA_VOID, NULL)` will print the line

```
0: MyFunc                                : Entered
```

when the debug level of 12 is specified. `PGADebugPrint(ctx, PGA_DEBUG_PRINTVAR, "MyFunc", "i = ", PGA_INT, (void *) &i)` will print the line

```
0: MyFunc                                : i = 1
```

when the debug level of 82 is specified. Users can also use the reserved debug levels of 1–10 to customize the trace facilities for use in their own functions. For example `PGADebugPrint(ctx, 5, "MyFunc", "After call to MyCleanUp", PGA_VOID, NULL);` will print the line

```

0: PGACreate                : Entered
0: PGASetRandomSeed         : Entered
0: PGASetMaxGAIterValue     : Entered
0: PGASetUp                 : Entered
0: PGACreatePop             : Entered
0: PGACreateIndividual      : Entered
:
:
0: PGACreateIndividual      : Entered
0: PGACreatePop             : Entered
0: PGACreateIndividual      : Entered
:
:
0: PGARun                   : Entered
0: PGARunSeq                : Entered
0: PGAEvaluate              : Entered
0: PGAFitness               : Entered
0: PGAGetStringLength       : Entered
:
:

```

Figure 11.1: PGAPack Partial Trace Output for Maxbit Example

```

0: MyFunc                    : After call to MyCleanUp

```

when the debug level of five is specified.

Note that we use `MPI_COMM_WORLD` (1) for the random number seed and (2) for `PGADebugPrint` calls.

Table 11.1: Debug Levels in PGAPack

0	Trace all debug prints
11	Trace high-level functions
12	Trace all function entries
13	Trace all function exits
20	Trace high-level parallel functions
21	Trace all parallel functions
22	Trace all send calls
23	Trace all receive calls
30	Trace Binary functions
32	Trace Integer functions
34	Trace Real functions
36	Trace Character functions
40	Trace population creation functions
42	Trace select functions
44	Trace mutation functions
46	Trace crossover functions
48	Trace function evaluation functions
50	Trace fitness calculation functions
52	Trace duplicate checking functions
54	Trace restart functions
56	Trace reporting functions
58	Trace stopping functions
60	Trace sorting functions
62	Trace random number functions
64	Trace system routines
66	Trace utility functions
80	Trace memory allocations
82	Trace variable print statements

Part III

Appendixes

Appendix A

Default Values

CONCEPT	DEFAULT	SET WITH
Population size	100	PGASetPopSize
Copied for population replacement	PGA_POPREPL_BEST	PGASetPopReplaceType
Stopping rule	PGA_STOP_MAXITER	PGASetStoppingRuleType
Maximum iterations	1000	PGASetMaxGAIterValue
Maximum no change iters	100	PGASetMaxNoChangeValue
Max. population homogeneity before stopping	95	PGASetMaxSimilarityValue
Number of new strings to generate	10	PGASetNumReplaceValue
Apply mutation and crossover	PGA_MIX_MUTATE_AND_CROSS	PGASetMixingType
Apply mutation or crossover	PGA_MIX_MUTATE_OR_CROSS	PGASetMixingType
Apply mutation only	PGA_MIX_MUTATE_ONLY	PGASetMixingType
Traditional crossover then mutation	PGA_MIX_TRADITIONAL	PGASetMixingType
Crossover type	PGA_CROSSOVER_TWOPT	PGASetCrossoverType
Probability of crossover	0.85	PGASetCrossoverProb
Uniform crossover bias	0.6	PGASetUniformCrossoverProb
Mutation type (Real strings)	PGA_MUTATION_GAUSSIAN	PGASetMutationType
Mutation type (Integer strings)	PGA_MUTATION_PERMUTE	PGASetMutationType
Mutation type (Character strings)	Same as initialization	PGASetCharacterInitType
Mutation probability	1/L	PGASetMutationProb
Real mutation constant	0.1	PGASetMutationRealValue
Integer mutation constant	1	PGASetMutationIntegerValue
Mutation range bounded	PGA_TRUE	PGASetMutationBoundedFlag
Select type	PGA_SELECT_TOURNAMENT	PGASetSelectType
Probabilistic binary tournament parameter	0.6	PGASetPTournamentProb
Use restart operator	PGA_FALSE	PGASetRestartFlag
Restart frequency	50	PGASetRestartFrequencyValue
Restart allele mutation rate	0.5	PGASetRestartAlleleChangeProb
Allow duplicate strings	PGA_FALSE	PGASetNoDuplicatesFlag
Fitness type	PGA_FITNESS_RAW	PGASetFitnessType
Fitness type for minimization	PGA_FITNESSMIN_CMAX	PGASetFitnessMinType
Multiplier for minimization problems	1.01	PGASetCMaxValue
Parameter MAX in fitness by ranking	1.2	PGASetMaxFitnessRank
Frequency of statistics printing	10	PGASetPrintFrequencyValue
L is the string length		

Table A.1: PGAPack Default Values

CONCEPT	DEFAULT	SET WITH
Print strings	PGA_FALSE	PGASetPrintOptions
Print offline statistics	PGA_FALSE	PGASetPrintOptions
Print online statistics	PGA_FALSE	PGASetPrintOptions
Print best string	PGA_FALSE	PGASetPrintOptions
Print worst string	PGA_FALSE	PGASetPrintOptions
Print Hamming distance	PGA_FALSE	PGASetPrintOptions
Randomly initialize population	PGA_TRUE	PGASetRandomInitFlag
Probability of initializing a bit to one	0.5	PGASetBinaryInitProb
How to initialize real strings	Range	PGASetrealInitRange
Real initialization range	$[0, 1]$	PGASetRealInitRange
How to initialize integer strings	Permutation	PGASetIntegerInitPermute
Integer initialization range	$[0, L - 1]$	PGASetIntegerInitPermute
How to initialize character strings	PGA_CINIT_LOWER	PGASetCharacterInitFlag
Seed random number with clock	PGA_TRUE	PGASetRandomSeed
Default MPI communicator	MPI_COMM_WORLD	PGASetCommunicator
Differential Evolution (DE) variant	PGA_DE_VARIANT_RAND	PGASetDEVariant
DE scale factor F	0.9	PGASetDEScaleFactor
DE auxiliary factor K	$0.5 \cdot (F + 1)$	PGASetDEAuxFactor
DE Crossover prob Cr	0.9	PGASetDECrossoverProb
DE Jitter	0.0	PGASetDEJitter
DE Either/Or Probability	0.5	PGASetDEProbabilityEO
DE Number of differences	1	PGASetDENumDiffs

L is the string length

Table A.2: PGAPack Default Values (continued)

Appendix B

Function Bindings

Symbolic Constants

PGAPack defines many symbolic constants that are used as arguments to PGAPack functions. These constants are the same for both Fortran and C. Below is a list of these constants. These constants are the same for both Fortran and C.

- PGAPack Data Types
 - PGA_DATATYPE_BINARY
 - PGA_DATATYPE_INTEGER
 - PGA_DATATYPE_REAL
 - PGA_DATATYPE_CHARACTER
 - PGA_DATATYPE_USER
- String Types
 - PGABinary
 - PGAInteger
 - PGAReal
 - PGACcharacter
- Data Types used in PGAError Calls
 - PGA_INT
 - PGA_DOUBLE
 - PGA_CHAR
 - PGA_VOID
- True and False
 - PGA_TRUE
 - PGA_FALSE
- Miscellaneous PGAPack Flags
 - PGA_FATAL
 - PGA_WARNING
 - PGA_UNINITIALIZED_INT

- PGA_UNINITIALIZED_DOUBLE
- PGAPack Temporary and Population Constants
 - PGA_TEMP1
 - PGA_TEMP2
 - PGA_OLDPOP
 - PGA_NEWPOP
- Debug Levels
 - PGA_DEBUG_ENTERED
 - PGA_DEBUG_EXIT
 - PGA_DEBUG_MALLOC
 - PGA_DEBUG_PRINTVAR
 - PGA_DEBUG_SEND
 - PGA_DEBUG_RECV
- Direction of Optimization
 - PGA_MAXIMIZE
 - PGA_MINIMIZE
- Stopping Criteria
 - PGA_STOP_MAXITER
 - PGA_STOP_NOCHANGE
 - PGA_STOP_TOOSIMILAR
- Crossover
 - PGA_CROSSOVER_ONEPT
 - PGA_CROSSOVER_TWOPT
 - PGA_CROSSOVER_UNIFORM
- Fitness
 - PGA_FITNESS_RAW
 - PGA_FITNESS_NORMAL
 - PGA_FITNESS_RANKING
- Fitness Minimization
 - PGA_FITNESSMIN_RECIPROCAL
 - PGA_FITNESSMIN_CMAX
- Mutation Type
 - PGA_MUTATION_CONSTANT
 - PGA_MUTATION_RANGE
 - PGA_MUTATION_UNIFORM
 - PGA_MUTATION_GAUSSIAN
 - PGA_MUTATION_PERMUTE

- Population Replacement
 - PGA_POPREPL_BEST
 - PGA_POPREPL_RANDOM_NOREP
 - PGA_POPREPL_RANDOM_REP
 - PGA_POPREPL_RTR
- Initialization Options
 - PGA_CINIT_LOWER
 - PGA_CINIT_UPPER
 - PGA_CINIT_MIXED
 - PGA_IINIT_PERMUTE
 - PGA_IINIT_RANGE
 - PGA_RINIT_PERCENT
 - PGA_RINIT_RANGE
- Report Options
 - PGA_REPORT_ONLINE
 - PGA_REPORT_OFFLINE
 - PGA_REPORT_HAMMING
 - PGA_REPORT_STRING
 - PGA_REPORT_WORST
 - PGA_REPORT_AVERAGE
- Selection
 - PGA_SELECT_PROPORTIONAL
 - PGA_SELECT_SUS
 - PGA_SELECT_TOURNAMENT
 - PGA_SELECT_PTournament
 - PGA_SELECT_TRUNCATION
- User Functions
 - PGA_USERFUNCTION_CREATESTRING
 - PGA_USERFUNCTION_MUTATION
 - PGA_USERFUNCTION_CROSSOVER
 - PGA_USERFUNCTION_PRINTSTRING
 - PGA_USERFUNCTION_COPYSTRING
 - PGA_USERFUNCTION_DUPLICATE
 - PGA_USERFUNCTION_INITSTRING
 - PGA_USERFUNCTION_BUILDDATATYPE
 - PGA_USERFUNCTION_STOPCOND
 - PGA_USERFUNCTION_ENDOFGEN
 - PGA_USERFUNCTION_GEN_DIFFERENCE
 - PGA_USERFUNCTION_PRE_EVAL

ANSI C Bindings

The use of any PGAPack function requires that the user have `#include "pgapack.h"` at the top of the file that references PGAPack functions.

Type	Function
MPI_Datatype	PGABuildDatatype(PGAContext *ctx, int p, int pop)
void	PGACHange(PGAContext *ctx, int p, int pop)
int	PGACheckStoppingConditions(PGAContext *ctx)
int	PGACheckSum(PGAContext *ctx, int p, int pop)
void	PGAClearDebugLevel(PGAContext *ctx, int level)
void	PGAClearDebugLevelByName(PGAContext *ctx, char *funcname)
void	PGACopyIndividual(PGAContext *ctx, int p1, int pop1, int p2, int pop2)
PGAContext*	PGACreate(int *argc, char **argv, int datatype, int len, int maxormin)
void	PGACrossover(PGAContext *ctx, int p1, int p2, int pop1, int c1, int c2, int pop2)
void	PGADebugPrint(PGAContext *ctx, int level, char *funcname, char *msg, int datatype, void *data)
void	PGADestroy(PGAContext *ctx)
int	PGADone(PGAContext *ctx, MPI_Comm comm)
int	PGADuplicate(PGAContext *ctx, int p, int pop1, int pop2, int n)
void	PGAEncodeIntegerAsBinary(PGAContext *ctx, int p, int pop, int start, int end, int val)
void	PGAEncodeIntegerAsGrayCode(PGAContext *ctx, int p, int pop, int start, int end, int val)
void	PGAEncodeRealAsBinary(PGAContext *ctx, int p, int pop, int start, int end, double low, double high, double val)
void	PGAEncodeRealAsGrayCode(PGAContext *ctx, int p, int pop, int start, int end, double low, double high, double val)
void	PGAError(PGAContext *ctx, char *msg, int level, int datatype, void *data)
void	PGAEvaluate(PGAContext *ctx, int pop, double(*f)(PGAContext *, int, int), MPI_Comm comm)
void	PGAFitness(PGAContext *ctx, int popindex)
int	PGAGetBestIndex(PGAContext *ctx, int pop)
int	PGAGetBinaryAllele(PGAContext *ctx, int p, int pop, int i)
double	PGAGetBinaryInitProb(PGAContext *ctx)
char	PGAGetCharacterAllele(PGAContext *ctx, int p, int pop, int i)
MPI_Comm	PGAGetCommunicator(PGAContext *ctx)
double	PGAGetCrossoverProb(PGAContext *ctx)
int	PGAGetCrossoverType(PGAContext *ctx)
int	PGAGetDataType(PGAContext *ctx)
double	PGAGetEvaluation(PGAContext *ctx, int p, int pop)
int	PGAGetEvaluationUpToDateFlag(PGAContext *ctx, int p, int pop)
double	PGAGetFitness(PGAContext *ctx, int p, int pop)
double	PGAGetFitnessCmaxValue(PGAContext *ctx)
int	PGAGetFitnessMinType(PGAContext *ctx)
int	PGAGetFitnessType(PGAContext *ctx)
int	PGAGetGAIterValue(PGAContext *ctx)
int	PGAGetIntegerAllele(PGAContext *ctx, int p, int pop, int i)
int	PGAGetIntegerFromBinary(PGAContext *ctx, int p, int pop, int start, int end)
int	PGAGetIntegerFromGrayCode(PGAContext *ctx, int p, int pop, int start, int end)
int	PGAGetIntegerInitType(PGAContext *ctx)

Type	Function
double	PGAGetMaxFitnessRank(PGAContext *ctx)
int	PGAGetMaxGAlterValue(PGAContext *ctx)
int	PGAGetMaxIntegerInitValue(PGAContext *ctx, int i)
double	PGAGetMaxMachineDoubleValue(PGAContext *ctx)
int	PGAGetMaxMachineIntValue(PGAContext *ctx)
double	PGAGetMaxRealInitValue(PGAContext *ctx, int i)
int	PGAGetMinIntegerInitValue(PGAContext *ctx, int i)
double	PGAGetMinMachineDoubleValue(PGAContext *ctx)
int	PGAGetMinMachineIntValue(PGAContext *ctx)
double	PGAGetMinRealInitValue(PGAContext *ctx, int i)
int	PGAGetMutationAndCrossoverFlag(PGAContext *ctx)
int	PGAGetMutationBoundedFlag(PGAContext *ctx)
int	PGAGetMutationIntegerValue(PGAContext *ctx)
int	PGAGetMutationOnlyFlag(PGAContext *ctx)
int	PGAGetMutationOrCrossoverFlag(PGAContext *ctx)
double	PGAGetMutationProb(PGAContext *ctx)
double	PGAGetMutationRealValue(PGAContext *ctx)
int	PGAGetMutationType(PGAContext *ctx)
int	PGAGetNoDuplicatesFlag(PGAContext *ctx)
int	PGAGetNumProcs(PGAContext *ctx, MPI_Comm comm)
int	PGAGetNumReplaceValue(PGAContext *ctx)
int	PGAGetOptDirFlag(PGAContext *ctx)
double	PGAGetPTournamentProb(PGAContext *ctx)
int	PGAGetPopReplaceType(PGAContext *ctx)
int	PGAGetPopSize(PGAContext *ctx)
int	PGAGetPrintFrequencyValue(PGAContext *ctx)
int	PGAGetRandomInitFlag(PGAContext *ctx)
int	PGAGetRandomizeSelect(PGAContext *ctx)
int	PGAGetRandomSeed(PGAContext *ctx)
int	PGAGetRank(PGAContext *ctx, MPI_Comm comm)
double	PGAGetRealAllele(PGAContext *ctx, int p, int pop, int i)
double	PGAGetRealFromBinary(PGAContext *ctx, int p, int pop, int start, int end, double lower, double upper)
double	PGAGetRealFromGrayCode(PGAContext *ctx, int p, int pop, int start, int end, double lower, double upper)
int	PGAGetRealInitType(PGAContext *ctx)
double	PGAGetRestartAlleleChangeProb(PGAContext *ctx)
int	PGAGetRestartFlag(PGAContext *ctx)
int	PGAGetRestartFrequencyValue(PGAContext *ctx)
int	PGAGetRTRWindowSize(PGAContext *ctx)
int	PGAGetSelectType(PGAContext *ctx)
int	PGAGetSortedPopIndex(PGAContext *ctx, int n)
int	PGAGetStoppingRuleType(PGAContext *ctx)
int	PGAGetStringLength(PGAContext *ctx)
int	PGAGetTournamentSize(PGAContext *ctx)
int	PGAGetTournamentWithReplacement(PGAContext *ctx)
double	PGAGetTruncationProportion(PGAContext *ctx)
double	PGAGetUniformCrossoverProb(PGAContext *ctx)
int	PGAGetWorstIndex(PGAContext *ctx, int pop)
double	PGAHammingDistance(PGAContext *ctx, int popindex)
double	PGAMean(PGAContext *ctx, double *a, int n)
int	PGAMutate(PGAContext *ctx, int p, int pop)

Type	Function
void	PGAPrintContextVariable(PGAContext *ctx, FILE *fp)
void	PGAPrintIndividual(PGAContext *ctx, FILE *fp, int p, int pop)
void	PGAPrintPopulation(PGAContext *ctx, FILE *fp, int pop)
void	PGAPrintReport(PGAContext *ctx, FILE *fp, int pop)
void	PGAPrintString(PGAContext *ctx, FILE *file, int p, int pop)
void	PGAPrintVersionNumber(PGAContext *ctx)
double	PGARandom01(PGAContext *ctx, int newseed)
int	PGARandomFlip(PGAContext *ctx, double p)
double	PGARandomGaussian(PGAContext *ctx, double mean, double sigma)
int	PGARandomInterval(PGAContext *ctx, int start, int end)
double	PGARandomUniform(PGAContext *ctx, double start, double end)
int	PGARank(PGAContext *ctx, int p, int *order, int n)
void	PGAReceiveIndividual(PGAContext *ctx, int p, int pop, int source, int tag, MPI_Comm comm, MPI_Status *status)
void	PGARestart(PGAContext *ctx, int source_pop, int dest_pop)
int	PGARound(PGAContext *ctx, double x)
void	PGARun(PGAContext *ctx, double(*evaluate)(PGAContext *c, int p, int pop, double *aux))
void	PGARunGM(PGAContext *ctx, double(*f)(PGAContext *, int, int), MPI_Comm comm)
void	PGARunMutationAndCrossover(PGAContext *ctx, int oldpop, int newpop)
void	PGARunMutationOnly(PGAContext *ctx, int oldpop, int newpop)
void	PGARunMutationOrCrossover(PGAContext *ctx, int oldpop, int newpop)
void	PGASelect(PGAContext *ctx, int popix)
int	PGASelectNextIndex(PGAContext *ctx, int pop)
void	PGASendIndividual(PGAContext *ctx, int p, int pop, int dest, int tag, MPI_Comm comm)
void	PGASendReceiveIndividual(PGAContext *ctx, int send_p, int send_pop, int dest, int send_tag, int recv_p, int recv_pop, int source, int recv_tag, MPI_Comm comm, MPI_Status *status)
void	PGASetBinaryAllele(PGAContext *ctx, int p, int pop, int i, int val)
void	PGASetBinaryInitProb(PGAContext *ctx, double probability)
void	PGASetCharacterAllele(PGAContext *ctx, int p, int pop, int i, char value)
void	PGASetCharacterInitType(PGAContext *ctx, int value)
void	PGASetCommunicator(PGAContext *ctx, MPI_Comm comm)
void	PGASetCrossoverProb(PGAContext *ctx, double crossover_prob)
void	PGASetCrossoverType(PGAContext *ctx, int crossover_type)
void	PGASetDebugLevel(PGAContext *ctx, int level)
void	PGASetDebugLevelByName(PGAContext *ctx, char *funcname)
void	PGASetEvaluation(PGAContext *ctx, int p, int pop, double val)
void	PGASetEvaluationUpToDateFlag(PGAContext *ctx, int p, int pop, int status)
void	PGASetFitnessCmaxValue(PGAContext *ctx, double val)
void	PGASetFitnessMinType(PGAContext *ctx, int fitness_type)
void	PGASetFitnessType(PGAContext *ctx, int fitness_type)
void	PGASetIntegerAllele(PGAContext *ctx, int p, int pop, int i, int value)
void	PGASetIntegerInitPermute(PGAContext *ctx, int min, int max)
void	PGASetIntegerInitRange(PGAContext *ctx, int *min, int *max)
void	PGASetMaxFitnessRank(PGAContext *ctx, double fitness_rank_max)
void	PGASetMaxGAIterValue(PGAContext *ctx, int maxiter)
void	PGASetMaxNoChangeValue(PGAContext *ctx, int max_no_change)
void	PGASetMaxSimilarityValue(PGAContext *ctx, int max_similarity)

Type	Function
void	PGASetMixingType(PGAContext *ctx, int type)
void	PGASetMutationBoundedFlag(PGAContext *ctx, int val)
void	PGASetMutationIntegerValue(PGAContext *ctx, int val)
void	PGASetMutationProb(PGAContext *ctx, double mutation_prob)
void	PGASetMutationRealValue(PGAContext *ctx, double val)
void	PGASetMutationType(PGAContext *ctx, int mutation_type)
void	PGASetNoDuplicatesFlag(PGAContext *ctx, int no_dup)
void	PGASetNumReplaceValue(PGAContext *ctx, int pop_replace)
void	PGASetPTournamentProb(PGAContext *ctx, double ptournament_prob)
void	PGASetPopReplaceType(PGAContext *ctx, int pop_replace)
void	PGASetPopSize(PGAContext *ctx, int popsize)
void	PGASetPrintFrequencyValue(PGAContext *ctx, int print_freq)
void	PGASetPrintOptions(PGAContext *ctx, int option)
void	PGASetRandomInitFlag(PGAContext *ctx, int RandomBoolean)
void	PGASetRandomizeSelect(PGAContext *ctx, int value)
void	PGASetRandomSeed(PGAContext *ctx, int seed)
void	PGASetRealAllele(PGAContext *ctx, int p, int pop, int i, double value)
void	PGASetRealInitPercent(PGAContext *ctx, double *median, double *percent)
void	PGASetRealInitRange(PGAContext *ctx, double *min, double *max)
void	PGASetRestartAlleleChangeProb(PGAContext *ctx, double prob)
void	PGASetRestartFlag(PGAContext *ctx, int val)
void	PGASetRestartFrequencyValue(PGAContext *ctx, int numiter)
void	PGASetRTRWindowSize(PGAContext *ctx, int size)
void	PGASetSelectType(PGAContext *ctx, int select_type)
void	PGASetStoppingRuleType(PGAContext *ctx, int stoprule)
void	PGASetTournamentSize(PGAContext *ctx, int size)
void	PGASetTournamentWithReplacement(PGAContext *ctx, int v)
void	PGASetTruncationProportion(PGAContext *ctx, double prop)
void	PGASetUniformCrossoverProb(PGAContext *ctx, double uniform_cross_prob)
void	PGASetUp(PGAContext *ctx)
void	PGASetUserFunction(PGAContext *ctx, int constant, void *f)
void	PGASortPop(PGAContext *ctx, int pop)
double	PGAStddev(PGAContext *ctx, double *a, int n, double mean)
void	PGAUpdateGeneration(PGAContext *ctx, MPI_Comm comm)
void	PGAUsage(PGAContext *ctx)

Fortran 77 Bindings

Use the rules defined in Chapter 10 (and the machine-specific idiosyncrasies noted in Appendix D) to determine the Fortran bindings.

Appendix C

Parallelism Background

Parallel Computer Taxonomy

Traditionally, parallel computers are classified according to Flynn's taxonomy [12]. Flynn's classification distinguishes parallel computers according to the number of instruction streams and data operands being computed on simultaneously.

Flynn's single-instruction single-data (SISD) model is the traditional sequential computer. A single program counter fetches instructions from memory. The instructions are executed on *scalar* operands. There is no parallelism in this model.

In the single-instruction multiple-data (SIMD) model there is again a single program counter fetching instructions from memory. However, now the operands of the instructions can be one of two types: either scalar or array. If the instruction calls for execution involving only scalar operands, it is executed by the control processor (i.e., the central processing unit fetching instructions from memory). If, on the other hand, the instruction calls for execution using array operands, it is broadcast to the *array* of processing elements. The processing elements are separate computing devices that rely upon the control processor to determine the instructions they will execute.

In a multiple-instruction multiple-data (MIMD) computer there exist multiple processors each of which has its own program counter. Processors execute independently of each other according to whatever instruction the program counter points to next. MIMD computers are usually further subdivided according to whether the processors share memory or each has its own memory.

In a shared-memory MIMD computer both the program's instructions and the part of the program's data to be shared exist within a single shared memory. Additionally, some data may be private to a processor and not be globally accessible by other processors. The processors execute asynchronously of each other. Communication and synchronization between the processors are handled by having them each read or write a shared-memory location.

A distributed-memory MIMD computer consists of multiple "nodes." A node consists of a processor, its own memory, a network interface, and sometimes a local disk. The program instructions and data reside in the node's memory. The nodes are connected via some type of network that allows them to communicate with each other. Parallelism is achieved by having each processor compute simultaneously on the data in its own memory. Communication and synchronization are handled by passing of messages (a destination node address and the local data to be sent) over the interconnection network.

Processes vs. Processors

We distinguish the two terms process and processor. A *process* is a software abstraction with a unique address space that can be scheduled by the operating system. A *processor* is the physical computer hardware on which computations take place.

On MIMD parallel computers, usually one process executes on each processor (although this is not required). On a uniprocessor, multiple processes timeshare the single processor.

Message-Passing Programming Model

In the message-passing programming model multiple processes communicate by passing messages—transferring data from the address space of one process into the address space of another process. When a process needs data stored in the memory of another process, the data must be sent from the process that “owns” it, to the memory of the process that needs it.

The message-passing programming model is currently one of the most popular. One reason for the popularity is portability. Message passing is the natural programming model on distributed-memory MIMD computers. Each process is naturally mapped to one of the machine’s nodes. A similar implementation is common on a workstation network where one process runs on each workstation. On a shared-memory MIMD computer multiple processes can emulate message passing by communicating only via logical message queues—areas of shared memory partitioned by process. On a uniprocessor the multiple processes that timeshare the physical processor can also emulate the idea of logical message queues for their communication.

One example of the message-passing programming model is the master/slave model. In this model a *master* process distributed work (computation to be performed) to the slave processes. The slaves perform the work and return the result to the master. In many implementations the master plays a bookkeeping role only and does not perform any computation.

Parallel Genetic Algorithms

When using the term “parallel genetic algorithm” it is important to distinguish between parallel models, their (parallel or sequential) implementation, and the computer hardware.

Models

A sequential GA model (more accurately called a *global* model) has a single population and no restrictions (partitioning) upon which strings recombine with which. The sequential GA is the traditional GA model given in the literature. In a parallel GA model there are either multiple populations (an island model) or a partitioning of a single population (often called a fine-grained model).

Implementations

Both parallel and sequential GA models can have parallel or sequential implementations. A sequential implementation of the global model is the traditional approach discussed in the GA literature. One process, running on a uniprocessor (PCs and workstations), performs all the calculations. In a parallel implementation of the global model the steps of the GA (some or all of selection, crossover, mutation, and fitness calculation) are executed simultaneously by multiple processes running on a parallel computer or workstation network.

In a sequential implementation of a parallel GA model, multiple processes, each responsible for a subpopulation or partition of the full population, time share the processor of the uniprocessor they are running on. In a parallel implementation of a parallel GA model, the multiple processes each run on a unique processor of a parallel computer or workstation network.

MPI

MPI (Message Passing Interface) is a *specification* of a message-passing library for parallel computers and workstation networks—it defines a set of functions and their behavior. The actual *implementation* of this interface is left up to vendors and researchers to develop. At the time of this writing several implementations of MPI, both proprietary and freely available, exist. MPI was designed by a large group of parallel computer vendors, computer researchers, and application developers as a standard for message passing.

Communicators

Almost all MPI functions require a *communicator*. If MPI routines are called directly, the user must supply a communicator. Also, if any of PGAPack's parallel routines, other than `PGARun`, are used, the user must supply a communicator as well.

A communicator combines the notions of context and group. A *context* is an extension of the notion of a "tag" used in many other message-passing systems to identify a message. Contexts differ from tags in that they are allocated by the system, not the user, and that no wild-card matching among contexts is allowed. A *group* contains n processes whose *rank* is an integer between $0, \dots, n - 1$. Processes may belong to more than one group and have a unique rank within each.

Any MPI implementation will always supply the default communicator `MPI_COMM_WORLD`. This communicator contains all processes that were created when MPI was initialized. For most users this is all they have to know about communicators. Simply supply `MPI_COMM_WORLD` whenever a communicator is required as an argument. For more sophisticated use, users are referred to [13, 15].

Parallel I/O

The issue of parallel I/O is independent of PGAPack. However, since we assume many PGAPack users will wish to do I/O we address this topic. A primary consideration has to do with whether one or all processors do I/O. Consider the following two code fragments, keeping in mind that they are being executed simultaneously by *multiple* processes:

```
ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 30, PGA_MINIMIZE)

and

int len;
scanf("%d",&len);
ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, len, PGA_MINIMIZE);
```

In the first case, all processes will receive the value of 30 for the string length since it is a constant. In the second case, however, the value of the string length is determined at run time. Whether one or all processes execute the `scanf` function is undefined in MPI and depends on the particular parallel computing environment. In PGAPack we require that all processes have the same values for all fields in the context variable. Since some of these fields may be set by using values specified at run time, we suggest that your I/O that reads in PGAPack parameters be done as follows:

```
#include "pgapack.h"
double evaluate (PGAContext *ctx, int p, int pop, double *aux);

int main( int argc, char **argv )
{
    PGAContext *ctx;
    int myid, len;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0) {
        printf("String length? ");
        scanf("%d", &len);
    }
    MPI_Bcast(&len, 1, MPI_INT, 0, MPI_COMM_WORLD);

    ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, len, PGA_MAXIMIZE);
    PGASetUp(ctx);
    PGARun(ctx, evaluate);
    PGADestroy(ctx);
}
```

```
    MPI_Finalize();  
    return(0);  
}
```

The key point is that *only* process 0 (as determined by `MPI_Comm_rank`) performs I/O and the value of `len` is then broadcast (using `MPI_Bcast`) to the other processes.

Appendix D

Machine Idiosyncrasies

Data Type Sizes

PGAPack is written entirely in ANSI C. However, because it is callable from Fortran, and no standards exist for interlanguage communication, problems may arise. These have to do with a lack of consistency in the size of data types between the two languages.

On all machines we have tested, an **integer** declaration in Fortran is the same size as an **int** declaration in C and everything works properly. For floating-point numbers, however, we have found at least one inconsistency. The requirement is for the Fortran floating-point number to be the same size as a C **double**. On most machines a Fortran **double precision** declaration is the equivalent size. On the Cray T3D, however, by default, the Fortran data type **double precision** is not supported and must be handled as described below.

Since Fortran does not support pointers, an **integer** variable is used to hold the address of the context variable (and possibly MPI communicator addresses as well). Therefore, a Fortran **integer** must be “large enough” to hold an address on the machine. For all 32-bit address space machines we have tested this is the case. On machines with a 64-bit address space, however, this may not be true. In particular, the size of a Fortran **integer** on the Silicon Graphics Power Challenge and DEC Alpha (but *not* the Cray T3D) is 32-bits and is not large enough to hold a machine address. The solution on these machines is to use the (nonstandard, but supported) Fortran declaration **integer*8** for the context variable.

Startup

The MPI standard provides for *source code* portability. However, the MPI standard does *not* specify how an MPI program shall be started or how the number of processes in the computation is specified. These will vary according to the computer being used and the choice of MPI implementation. The notes below are from our experiences testing PGAPack on different machines.

Silicon Graphics Challenge

The Silicon Graphics Challenge is a 32-bit symmetric multiprocessor. We used MPICH with the **ch_shmem** device and the **ncc** C compiler. Several warnings were received

```
warning(3262): parameter "ctx" declared and never referenced
warning(3141): cast between pointer-to-object and pointer-to-function
```

but the library was successfully built. To run a parallel PGAPack program, use either

```
a.out -np nprocs
```

or MPICH's **mpirun** command.

Silicon Graphics Power Challenge

The Silicon Graphics Power Challenge is similar to the Challenge, except that it has a 64-bit address space. On this machine the size of an integer (`int` in C and `integer` in Fortran) is not the same as the size of an address. Fortran users should use the declaration `integer*8` for the context variable (and `integer` for other Fortran integer declarations). See also Chapter 10.

We used MPICH with the `ch_p4` device and the the MIPSpro C compiler (`cc`). We found a bug in `pca`, the Power C Analyzer, and recommend not using it for now. (To do this do not specify the `-pca` switch to `cc`). To run a parallel PGAPack program, use

```
a.out -np nprocs
```

or MPICH's `mpirun` command.

Cray T3D

The Cray T3D has a 64-bit address space. However, the size of an integer on the T3D is the same as the size of an address, and therefore no special considerations are needed for declaring the context variable in Fortran.

On the T3D a C `double` is 64 bits. The Fortran `double precision` data type, however, is not supported (by default). One workaround is to declare all floating-point numbers `REAL`, as these are 64 bits on the T3D. The other workaround is to use the compiler switch `"-dp"`.

To compile for a Cray T3D, cross compilation is done on a front-end machine (a Cray C90 in our case). Set Cray's `TARGET` environment variable so the compiler, linker, etc., will know which architecture to compile for.

```
setenv TARGET cray-t3d
```

An alternative is to use `"-T cray-t3d"` with `cc` and `"-C cray-t3d"` with `cf77`. Another alternative is to explicitly use the cross compilers (`/mpp/bin/cc` and `/mpp/bin/cf77`) and linker (`/mpp/bin/mppldr`).

We used the MPI in `/usr/local/mpp/lib/libmpi.a`. Adding `-lmpi` in your link step may also find the MPI library. If a successful T3D executable was built, the command `"file a.out"` should say `"MPP absolute."`

To run a parallel PGAPack program, use

```
a.out -npes nprocs
```

where `nprocs` is a power of two.

Intel Paragon

We used MPICH with the `ch_nx` device and compiled with `cc -nx`. To run a parallel PGAPack program, use

```
a.out -sz nprocs
```

or MPICH's `mpirun` command.

IBM SP2

We tested the IBM SP2 using both MPICH with the `ch_eui` device, and IBM's research MPI, MPI-F. We compiled PGAPack with `xlc` and linked with `mpCC`. Execution required setting a number of environment variables. We were successful with the following, but this may vary with the system software installed on the SP you are using.

```
setenv MP_HOSTFILE /sphome/hostfile
setenv MP_PROCS      np
setenv MP_EUILIB     us
setenv MP_INFOLEVEL  0
setenv MP_HOLD_STDIN YES
setenv MP_PULSE      0
a.out
```

Convex Exemplar

We used MPICH with the `ch_shmem` device. Be sure to compile (the Fortran examples) with `fort77`, not `f77`. Also, you must link with `/usr/lib/libU77.a` *last* to satisfy `iargc` and `getarg`. This *must* be done *manually* in the prototype makefiles `./examples/fortran/Makefile.in` and `./examples/mgh/Makefile.in` *before* running `configure`. To run a parallel PGAPack program using MPICH use the `mpirun` command.

Sun SparcStation

We used MPICH with the `ch_p4` device and the GNU C compiler `gcc`. The `instverf` test program was run using 4 processes with:

```
/usr/local/mpi/bin/mpirun instverf -arch sun4 -np 4
```

Silicon Graphics Workstation

We used MPICH with the `ch_p4` device and `mpirun` command, the `cc` C compiler, and `f77` Fortran compiler.

IBM/RS6000 Workstation

We have successfully run PGAPack on both single workstations and networks of workstations using the MPICH implementation with the `ch_p4` device.

Hewlett Packard Workstation

We used MPICH with the `ch_shmem` device and `mpirun` command, the `gcc` C compiler, and `fort77` Fortran compiler.

DEC Alpha Workstation

DEC Alpha workstations have a 64-bit address space. On this machine the size of an integer (`int` in C and `integer` in Fortran) is not the same as the size of an address. Fortran users should use the declaration `integer*8` for the context variable (and `integer` for other Fortran integer declarations). See also Chapter 10.

Appendix E

Common Problems

- When reading input value to be used as parameters in **PGASet** calls, the **PGASet** calls themselves may not be executed until *after* **PGACreate** has been called.
- In C, when reading input parameters which are of type **double**, the **scanf** conversion specification should be of the form **%lf**, *not* **%f** which is appropriate for **floats**.
- An infinite loop can occur if the number of permutations of the bit string is less than the population size. For example, for a binary-valued string of length four, there are $2^4 = 16$ possibilities. If the population size is greater than 16, and duplicate strings are not allowed in the population, an infinite loop will occur.
- Erroneous results can occur if the name of a user's function conflicts with a library function used by PGAPack. For example, if a program defined its own **ceil** function, this would conflict with the C math library function of the same name.
- All floating point constants and variables used in PGAPack are of type **double**. Particularly from Fortran, the user should be careful to make sure that they pass a **double precision** constant or variable.
- **PGACreate** removes command line arguments. One consequence is that if **PGACreate** is called twice in the same program (unusual, but legal), the second **PGACreate** call will *not* receive the command-line arguments.
- If one includes **mpi.h** (or **mpif.h**) when it should not be, errors will result, as well as warnings about redefining macros and typedefs. This usually happens when a sequential version of PGAPack is used (with “fake” MPI stub routines and definitions) and the user's program explicitly includes “real” **mpi.h** or **mpif.h** header files.
- If one fails to include **mpi.h** (or **mpif.h**) when it should be (such as calling MPI functions directly) errors may result. Since **pgapack.h** includes **mpi.h** this should not happen in C. The Fortran include file, **pgapackf.h**, however, does *not* include **mpif.h**. The user must explicitly include it in every subroutine and function that makes MPI calls. Not including **mpif.h** could result in any of several different errors, including
 - syntax errors when compiling (for example, **MPI_COMM_WORLD** being undefined)
 - general errors in the computed results
 - the program crashing when it calls the undefined subroutine **MPI_Init**
 - general MPI errors such as:

```
0 - Error in MPI_COMM_RANK : Invalid communicator
[0] Aborting program!
```

We have also seen the following error from not including `bmplib.h` in the main program:

```
PGACreate: Invalid value of datatype: 0
PGAError: Fatal
```

- If the `ch_p4` device in MPICH is used to run on workstations one must have a correct processor group file (`procgroup`). The error message

```
(ptera-36%)a.out
p0_18429: p4_error: open error on procgroup file (procgroup): 0
(ptera-37%)
```

may occur if the processor group file is not specified correctly. See the MPICH users guide for more details.

- A common error with the `procgroup` file when using the `ch_p4` device in MPICH is to have an incorrect path to the executable.
- When compiling the `examples` directory we have seen “multiply defined” error messages. For example:

```
Making C examples
  Compiling classic
ld: /usr/local/mpi/lib/sun4/ch_p4/libmpi.a(initialize.o): _MPI_Initialized: multiply defined
collect2: ld returned 2 exit status
```

We have seen this error occur when a sequential version of PGAPack was built and the library (`./lib/arch/libpgag.a` or `./lib/arch/libpga0.a`) was not deleted before attempting to build a new, parallel version of PGAPack. The “fake” MPI stub routines are in the sequential library and have name conflicts when a “real” MPI library is referenced. The solution is to delete the old `.a` file and rerun `make install`.

Bibliography

- [1] J. Baker. Reducing bias and inefficiency in the selection algorithm. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 14–21, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates.
- [2] Julian Blank, Kalyanmoy Deb, and Proteek Chandan Roy. Investigating the normalization procedure of NSGA-III. In Kalyanmoy Deb, Erik Goodman, Carlos A. Coello Coello, Kathrin Klamroth, Kaisa Miettinen, Sanaz Mostaghim, and Patrick Reed, editors, *Evolutionary Multi-Criterion Optimization, 10th International Conference (EMO)*, volume 11411 of *Lecture Notes in Computer Science*, pages 229–240. Springer, East Lansing, MI, USA, March 2019.
- [3] Indraneel Das and J. E. Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM Journal on Optimization*, 8(3):631–657, August 1998.
- [4] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.
- [5] Kalyanmoy Deb. An efficient constraint handling method for genetic algorithms. *Computer Methods in Applied Mechanics and Engineering*, 186(2–4):311–338, June 2000.
- [6] Kalyanmoy Deb and Ram Bushan Agrawal. Simulated binary crossover for continuous search space. 9(2):115–148, 1995.
- [7] Kalyanmoy Deb and Debayan Deb. Analysing mutation schemes for real-parameter genetic algorithms. *International Journal of Artificial Intelligence and Soft Computing*, 4(1):1–28, February 2014.
- [8] Kalyanmoy Deb and Mayank Goyal. A combined genetic adaptive search (GeneAS) for engineering design. *Computer Science and Informatics*, 26(4):30–45, 1996.
- [9] Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, August 2014.
- [10] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, April 2002.
- [11] Vladimir Filipović, Jozef Kratica, Dušan Tošić, and Ivana Ljubić. Fine grained tournament selection for the simple plant location problem. In *5th Online World Conference on Soft Computing Methods in Industrial Applications*, pages 152–158. January 2000.
- [12] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.
- [13] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.
- [14] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989.

- [15] W. Gropp, E. Lusk, and A. Skjellum. *USING MPI Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, 1994.
- [16] Georges Harik. Finding multiple solutions in problems of bounded difficulty. IlliGAL Report 94002, Illinois Genetic Algorithm Lab, May 1994.
- [17] Georges R. Harik. Finding multimodal solutions using restricted tournament selection. In Larry J. Eshelman, editor, *Proceedings of the International Conference on Genetic Algorithms (ICGA)*, pages 24–31. Morgan Kaufmann, July 1995.
- [18] Georges R. Harik and David E. Goldberg. Learning linkage. In Richard K. Belew and Michael D. Vose, editors, *Foundations of Genetic Algorithms (FOGA) 4*, pages 247–262, San Diego, CA, August 1996. Morgan Kaufmann.
- [19] J. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, Cambridge, 1992.
- [20] Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part II: Handling constraints and extending to an adaptive approach. *IEEE Transactions on Evolutionary Computation*, 18(4):602–622, August 2014.
- [21] Martin Pelikan. *Hierarchical Bayesian Optimization Algorithm: Toward a New Generation of Evolutionary Algorithms*, volume 170 of *Studies in Fuzziness and Soft Computing*. Springer, 2005.
- [22] Kenneth V. Price, Rainer M. Storn, and Jouni A. Lampinen. *Differential Evolution: A Practical Approach to Global Optimization*. Springer, Berlin, Heidelberg, 2005.
- [23] Ralf Salomon. Re-evaluating genetic algorithm performance under coordinate rotation of benchmark functions. a survey of some theoretical and practical aspects of genetic algorithms. *Biosystems*, 39(3):263–278, 1996.
- [24] Artem Sokolov and Darrell Whitley. Unbiased tournament selection. In Hans-Georg Beyer, editor, *Genetic and Evolutionary Computation GECCO 2005*, page 1131–1138, Washington DC, June 2005. ACM.
- [25] W. Spears and K. DeJong. On the virtues of parameterized uniform crossover. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236, San Mateo, 1991. Morgan Kaufmann.
- [26] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, International Computer Science Institute (ICSI), March 1995.
- [27] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Global Optimization*, 11(4):341–359, December 1997.
- [28] G. Syswerda. Uniform crossover in genetic algorithms. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9, San Mateo, 1989. Morgan Kaufmann.
- [29] Tetsuyuki Takahama and Setsuko Sakai. Constrained optimization by the ϵ constrained differential evolution with gradient-based mutation and feasible elites. In *IEEE International Conference on Evolutionary Computation (CEC)*. Vancouver, BC, Canada, July 2006.
- [30] Tetsuyuki Takahama and Setsuko Sakai. Constrained optimization by the ϵ constrained differential evolution with an archive and gradient-based mutation. In *IEEE Congress on Evolutionary Computation (CEC)*, Barcelona, Spain, July 2010.
- [31] Ryoji Tanabe and Alex Fukunaga. Reevaluating exponential crossover in differential evolution. In Thomas Bartz-Beielstein, Jürgen Branke, Bogdan Filipič, and Jim Smith, editors, *Parallel Problem Solving from Nature – PPSN XIII*, volume 8672 of *Lecture Notes in Computer Science*, pages 201–210. Springer, Ljubljana, Slovenia, September 2014.

- [32] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, San Mateo, 1989. Morgan Kaufmann.
- [33] D. Whitley and J. Kauth. GENITOR: A different genetic algorithm. In *Rocky Mountain Conference on Artificial Intelligence*, pages 118–130, Denver, 1988.