Users Guide to the PGAPack Parallel Genetic Algorithm Library

by

David Levine

Mathematics and Computer Science Division

January 1996

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

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

0 Quick Start ................................................. 1

I Getting Started ........................................... 2

1 Introduction ................................................. 3

2 Installation .................................................. 4
   2.1 Obtaining PGAPack ........................................ 4
   2.2 Requirements ............................................. 4
   2.3 Structure of the Distribution Directory ................. 4
   2.4 Installation Instructions .................................. 5
   2.5 Installation Examples ..................................... 6
      2.5.1 Sequential Installation ................................. 6
      2.5.2 Parallel Installation .................................. 6
   2.6 Mailing Lists, Web Page, and Bug Reporting ............ 7

3 Examples ..................................................... 8
   3.1 Maxbit Problem in C ....................................... 8
   3.2 Maxbit Problem in Fortran ................................. 8
   3.3 Specifying Nondefault Values .............................. 10
   3.4 Parallel I/O ............................................... 10
   3.5 Compiling, Linking, and Execution ....................... 12

II Users Guide .................................................. 14

4 The Structure of PGAPack .................................... 15
   4.1 Native Data Types .......................................... 15
   4.2 Context Variable ........................................... 15
   4.3 Levels of Usage Available ................................ 15
   4.4 Function Call-Based Library ............................... 16
   4.5 Header File and Symbolic Constants ....................... 16
   4.6 Evaluation Function ....................................... 16
   4.7 Parallelism ................................................ 16
   4.8 Implementation ............................................ 17

5 Basic Usage ................................................... 18
   5.1 Required Functions ........................................ 18
   5.2 Population Replacement ................................... 19
   5.3 Stopping Criteria .......................................... 21
   5.4 Initialization .............................................. 21
   5.5 Selection .................................................. 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6 Crossover</td>
<td>22</td>
</tr>
<tr>
<td>5.7 Mutation</td>
<td>22</td>
</tr>
<tr>
<td>5.8 Restart</td>
<td>23</td>
</tr>
<tr>
<td>5.9 String Evaluation and Fitness</td>
<td>23</td>
</tr>
<tr>
<td>5.10 Accessing Allele Values</td>
<td>24</td>
</tr>
<tr>
<td>5.10.1 Representing an Integer with a Binary String</td>
<td>25</td>
</tr>
<tr>
<td>5.10.2 Representing a Real Value with a Binary String</td>
<td>25</td>
</tr>
<tr>
<td>5.10.3 Example</td>
<td>26</td>
</tr>
<tr>
<td>5.11 Report Options</td>
<td>27</td>
</tr>
<tr>
<td>5.12 Utility Functions</td>
<td>27</td>
</tr>
<tr>
<td>5.12.1 Random Numbers</td>
<td>27</td>
</tr>
<tr>
<td>5.12.2 Print Functions</td>
<td>27</td>
</tr>
<tr>
<td>5.12.3 Miscellaneous</td>
<td>27</td>
</tr>
<tr>
<td>5.13 Command-Line Arguments</td>
<td>28</td>
</tr>
<tr>
<td>6 Explicit Usage</td>
<td>29</td>
</tr>
<tr>
<td>6.1 Notation</td>
<td>29</td>
</tr>
<tr>
<td>6.2 Simple Sequential Example</td>
<td>29</td>
</tr>
<tr>
<td>6.3 Complex Example</td>
<td>30</td>
</tr>
<tr>
<td>6.4 Explicit PGAPack Functions</td>
<td>32</td>
</tr>
<tr>
<td>7 Custom Usage: Native Data Types</td>
<td>33</td>
</tr>
<tr>
<td>7.1 Basics</td>
<td>33</td>
</tr>
<tr>
<td>7.2 Example Problem: C</td>
<td>34</td>
</tr>
<tr>
<td>7.3 Example Problem: Fortran</td>
<td>34</td>
</tr>
<tr>
<td>8 Custom Usage: New Data Types</td>
<td>37</td>
</tr>
<tr>
<td>8.1 Basics</td>
<td>37</td>
</tr>
<tr>
<td>8.2 Example Problem</td>
<td>37</td>
</tr>
<tr>
<td>9 Hill-Climbing and Hybridization</td>
<td>44</td>
</tr>
<tr>
<td>10 Parallel Aspects</td>
<td>46</td>
</tr>
<tr>
<td>10.1 Basic Usage</td>
<td>46</td>
</tr>
<tr>
<td>10.2 Explicit Use</td>
<td>46</td>
</tr>
<tr>
<td>10.3 Example</td>
<td>47</td>
</tr>
<tr>
<td>10.4 Performance</td>
<td>47</td>
</tr>
<tr>
<td>11 Fortran Interface</td>
<td>49</td>
</tr>
<tr>
<td>12 Debugging Tools</td>
<td>51</td>
</tr>
<tr>
<td>III Appendixes</td>
<td>54</td>
</tr>
<tr>
<td>A Default Values</td>
<td>55</td>
</tr>
<tr>
<td>B Function Bindings</td>
<td>57</td>
</tr>
<tr>
<td>C Parallelism Background</td>
<td>64</td>
</tr>
<tr>
<td>D Machine Idiosyncrasies</td>
<td>68</td>
</tr>
<tr>
<td>E Common Problems</td>
<td>71</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>73</td>
</tr>
<tr>
<td>Bibliography</td>
<td>74</td>
</tr>
</tbody>
</table>
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 assume you have ftped the compressed tar file pgapack.tar.Z containing the distribution into /home/username. To build a sequential version of PGAPack for a Sun SparcStation in /usr/local/pga and run a test example, type

1. `uncompress /home/username/pgapack.tar.Z`
2. `mkdir /usr/local/pga`
3. `cd /usr/local/pga`
4. `tar xvf /home/username/pgapack.tar`
5. `configure -arch sun4`
6. `make install`
7. `/usr/local/pga/examples/c/maxbit`

To build an optimized (no built-in debugging capabilities), parallel version of PGAPack for an IBM SP parallel computer, using an MPI implementation with include files in /usr/local/mpi/include and library in /usr/local/mpi/lib, and run a test example using four processes, type

1. `uncompress /home/username/pgapack.tar.Z`
2. `mkdir /usr/local/pga`
3. `cd /usr/local/pga`
4. `tar xvf /home/username/pgapack.tar`
5. `configure -arch rs6000 \
   -mpiinc /usr/local/mpi/include -mpilib /usr/local/mpi/lib/libmpi.a`
6. `make install`
7. `mpirun -np 4 /usr/local/pga/examples/c/maxbit`

Step 7, the execution step, is completely dependent on the MPI implementation. This example uses the mpirun script that is distributed with the MPICH implementation [1]. Other MPI implementations may have other ways to specify the number of processes to use.

More details on the installation process and various options are given in Chapter 2. Chapter 3 (example problems) and Sections 5.1 (required functions) and 5.9 (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

Installation

2.1 Obtaining PGAPack

The complete distribution of PGAPack is available by anonymous ftp from ftp.mcs.anl.gov in the file pub/pgapack/pgapack.tar.Z. The distribution contains all source code, installation instructions, this users guide, and a collection of examples in C and Fortran. The current release of PGAPack is 1.0. You can check which version of PGAPack you have by running any C language PGAPack program with the command-line option -pgaversion.

2.2 Requirements

To compile PGAPack, you must have an ANSI C compiler that includes a full implementation of the Standard C library and related header files. If you wish only to build a sequential version of PGAPack this is all that is required. To build a parallel version, you must have an implementation of the Message Passing Interface (MPI) [5, 6] for the parallel computer or workstation network you are running on. If you do not have a native version of MPI for your computer, several machine-independent implementations are available. Most of the testing and development of PGAPack was done by using the MPICH implementation of MPI which is freely available [1].

2.3 Structure of the Distribution Directory

The PGAPack distribution contains the following files and subdirectories:

- CHANGES: Changes new to this release of PGAPack.
- COPYRIGHT: The usage terms.
- README: General instructions, including how to build and install PGAPack.
- configure.in: The "source code" for the configure script.
- configure: A Unix shell script that configures Makefile.in for a specific architecture.
- Makefile.in: Prototype makefile that is configured into the file Makefile for a specific architecture by configure.
- docs: The user manual and any other supporting files.
- examples: A directory containing C and Fortran examples.
- include: The PGAPack include directory.
2.4 Installation Instructions

When installing PGAPack you make two choices: whether to build a sequential (the default) or parallel version (see the flags -mpiinc and -mpilib below) and whether to build an optimized (the default) or debug version (the -debug flag). In broad outline, the installation steps are as follows.

1. Make a directory to install PGAPack in (mkdir /usr/local/pga).
2. Change directories to the directory created in the last step (cd /usr/local/pga).
5. Untar the uncompressed PGAPack tar file (tar xvf pgapack.tar).
6. Use configure to configure the makesfiles (configure -arch ARCH_TYPE)
   where ARCH_TYPE is one of sun4 for Sun SparStations workstations, next for NeXT workstations, rs6000 for IBM RS/6000 workstations, irix for Silicon Graphics workstations, hpux for Hewlett Packard workstations, alpha for DEC Alpha workstations, linux for machines running Linux, freebsd for machines running FreeBSD, generic for generic 32-bit machines, powerchallenge for the Silicon Graphics Power Challenge Array, challenge for the Silicon Graphics Challenge, t3d for the Cray T3D, sp2 for the IBM SP2, paragon for the Intel Paragon, or exemplar for the Convex Exemplar.

The full configure options are configure -arch ARCH_TYPE [-cc CC] [-cflags CFLAGS] [-f77 FC] [-fflags FFLAGS] [-debug] [-mpiinc MPI_INCLUDE_DIRECTORY] [-mpilib MPI_LIBRARY] [-help] where all parameters except -arch are optional and do the following:

- -cc: The name of the ANSI C compiler, cc by default.
- -cflags: Options passed to the C compiler.
- -f77: The name of the Fortran 77 compiler, f77 by default. (The Fortran compiler is used only to compile the Fortran examples in the ./examples/fortran directory.)
- -fflags: Options passed to the Fortran compiler.
- -debug: If specified, enables the debugging features (see Chapter 12) and compiles the source code with the -g flag. If this flag is not specified the debugging features are disabled, and the library is compiled with the -O flag
- -mpiinc: The directory where MPI include files are located.
- -mpilib: The full path to the MPI library.

If -mpiinc and -mpilib are specified, a parallel version of PGAPack will be built. If these flags are not specified, a sequential version of PGAPack will be built.

7. Execute the makefile (make install).
8. Add PGAPack's man pages to your man page path. (setenv MANPATH "$MANPATH":"/home/pgapack/man")
9. Execute a test problem
2.5 Installation Examples

These installation examples assume you have ftped the compressed tar file pgapack.tar.Z containing the distribution into /home/username.

2.5.1 Sequential Installation

To build a sequential version of PGAPack for a Sun SparcStation in /usr/local/pga and run a test example, type:

1. uncompress /home/username/pgapack.tar.Z
2. mkdir /usr/local/pga
3. cd /usr/local/pga
4. tar xvf /home/username/pgapack.tar
5. configure -arch sun4
6. make install
7. /usr/local/pga/examples/c/maxbit

2.5.2 Parallel Installation

To build an optimized (no built-in debugging capabilities), parallel version of PGAPack for an IBM SP parallel computer using an MPI implementation with include files in /usr/local/mpi/include and library in /usr/local/mpi/lib, and run a test example using four processes, type:

1. uncompress /home/username/pgapack.tar.Z
2. mkdir /usr/local/pga
3. cd /usr/local/pga
4. tar xvf /home/username/pgapack.tar
5. configure -arch rs6000 \ -mpiinc /usr/local/mpi/include -mpilib /usr/local/mpi/lib/libmpi.a
6. make install
7. mpirun -np 4 /usr/local/pga/examples/c/maxbit

Step 7, the execution step, is completely dependent on the MPI implementation. This example uses the mpirun script that is distributed with the MPICH implementation [1]. Other MPI implementations may have other ways to specify the number of processes to use.
2.6 Mailing Lists, Web Page, and Bug Reporting

To join the PGAPack mailing list to receive announcements of new versions, enhancements, and bug fixes, send electronic mail to pgapack@mcs.anl.gov. Bug reports should be sent to pgapack-bugs@mcs.anl.gov. The World Wide Web page for PGAPack is http://www.mcs.anl.gov/pgapack.html and contains up-to-date news and a list of bug reports.

When reporting a bug, please include as much information and documentation as possible. Helpful information would include PGAPack version number (-pgaversion), MPI implementation and version used, configuration options, type of computer system, problem description, and error message output. It is helpful if you put a PGAPrintContextVariable call before and after the PGASetUp call. Additionally, if possible, build a debug version of PGAPack and send "high-level" output from running your program with the trace facility enabled (Chapter 12).
Chapter 3

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 3.1 presents a simple PGAPack program in C whose structure is sufficient to solve many problems. Section 3.2 presents this same program in Fortran. Section 3.3 shows how to change default values in PGAPack. Section 3.4 contains an example that shows how keyboard input may be read in an MPI environment. Finally, Section 3.5 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.

3.1 Maxbit Problem in C

Figure 3.1 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. PGA_Destroy 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. PGAGetStringLength returns the string length. PGAGetBinary Allele returns the value of the ith bit of string p in population pop.

3.2 Maxbit Problem in Fortran

The Fortran Maxbit problem in Figure 3.2 is similar to the C version in Figure 3.1. The Fortran include file is pgapackf.h and should be included in every Fortran function or subroutine that makes PGAPack calls. 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. Note that evaluate 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.

\[\text{Since not all Fortran compilers support the }-I\text{ mechanism for specifying the include file search path, you will need to copy or set up a symbolic link to pgapackf.h from the directory you are compiling a Fortran program in.}\]
#include "pgapack.h"

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

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

double evaluate (PGAContext *ctx, int p, int pop)
{
    int i, nbits, stringlen;

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

Figure 3.1: 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(nbits)
return
end

Figure 3.2: PGAPack Fortran Program for the Maxbit Example
Figure 3.3: Specifying Nondefault Values

3.3 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 3.3 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 5.

3.4 Parallel I/O

The examples in Figures 3.4 (C) and 3.5 (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 communicator\(^2\). 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 [5, 6].

The C binding for MPI_Bcast is

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

and the Fortran binding

\(^2\)See Appendix C
#include "pgapack.h"

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

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) {
        printf("String length? "); /* Process 0 has a dialog */
        scanf("%d", &len);  /* with the user and */
        printf("Max iterations? "); /* broadcasts the user’s */
        scanf("%d", &maxiter); /* parameters to all */
        /* 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 3.4: PGAPack Maxbit Example in C with I/O
double precision evaluate
external evaluate

integer ctx, myid, len, maxiter, ierror

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

Process 0 has a dialog with the user and broadcasts the user's 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, ierr)
call MPI_Bcast(maxiter, 1, MPI_INT, 0, MPI_COMM_WORLD, ierr)

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(ierr)
stop
end

Figure 3.5: PGAPack Maxbit Example in Fortran with I/O

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 *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).

3.5 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.

How you execute your program will depend on whether a sequential or parallel version of PGAPack was built.
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 4

The Structure of PGAPack

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

4.1 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 7).

The binary (or bit) data type (i.e., 11011111) 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., 1319241) is often used in routing and scheduling problems. The real-valued data type (i.e., 42716308) is useful in numerical optimization applications. The character-valued data type (i.e., hello11o wor1l) is useful for symbolic applications.

4.2 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.

4.3 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.

4.4 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 PAGAFitness.

- 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.

4.5 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.

4.6 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 5.9.

4.7 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 10).
4.8 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 [5, 6]. 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 5

Basic Usage

As the examples in Chapter 3 show, a PGPack program can be written with just four function calls and a string evaluation function. This basic usage is discussed further in Section 5.1. Sections 5.3–5.12 explain options available in PGPack. Section 5.13 discusses PGPack command line arguments.

5.1 Required Functions

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

```c
PGACreate (&argc, argv, datatype, len, maxormin);
```

PGACreate allocates space for the context variable, ctx (Section 4.2), and returns its address. argc and argv are the standard list of arguments to a C program. datatype must be one of PGPADATATYPE_BINARY, PGPADATATYPE_INTEGER, PGPADATATYPE_REAL, or PGPADATATYPE_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 PGPAMAXIMIZE or PGPAMINIMIZE to indicate whether the user's problem is maximization or minimization, respectively.

In Fortran the call to PGACreate is

```fortran
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,

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

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 PGPack 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);

and in Fortran

doUBLE PRECISION FUNCTION evaluate (ctx, p, pop)
INTEGER ctx, p, pop

The user must write the evaluation function, and it must have the calling sequence shown above and discussed further in Section 5.9. After PGARun terminates, PGADestroy is called to release all memory allocated by PGAPack.¹

Except for writing an evaluation function (Section 5.9) 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.

### 5.2 Population Replacement

Two population replacement schemes are common in the literature. The first, the generational replacement genetic algorithm (GRGA), replaces the entire population each generation and is the traditional genetic algorithm [7]. The second, the steady-state genetic algorithm (SSGA), typically replaces only a few strings each generation and is a more recent development [9, 10, 11]. 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);
PGASetPopReplacementType(ctx,P GA_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 PGASetPopReplacementType to PGA_POPREPL_RANDOM_REP or PGA_POPREPL_RANDOM_NOREP, respectively.

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 [3] 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 PGASetMutationOrCrossoverFlag (ctx,P GA_TRUE). To have strings undergo both crossover and mutation, one should use PGASetMutationAndCrossoverFlag (ctx,P GA_TRUE).

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,P GA_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 5.1 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 - PGAGetNumReplaceValue 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 PGASetPopReplacementType. The remaining PGAGetNumReplaceValue strings are created by crossover and mutation.

¹PGADestroy will also call MPI.finalize, if MPI was started by PGACreate.
Figure 5.1: Population Replacement
5.3 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.

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 or-ed 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.

5.4 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 \([0, \text{string length} - 1]\), 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

```c
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

```c
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,

```c
stringlen = PGAGetStringLength(ctx);
for (i = 0; i < stringlen; i++) {
    median[i] = (double) i;
}
```
percent[i] = .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, than 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.

5.5 Selection

The selection phase allocates reproductive trials to strings on the basis of their fitness. PGAPack supports four selection schemes: proportional selection, stochastic universal selection, binary tournament selection, and probabilistic binary tournament selection. The choice may be specified by setting the second argument of PGASetSelectType to one of PGA_SELECT_PROPORTIONAL, PGA_SELECT_SUS, PGA_SELECT_TOURNAMENT, and PGA_SELECT_PROB_TOURNAMENT for proportional, stochastic universal, tournament, and probabilistic tournament selection, respectively. The default is tournament selection. For probabilistic tournament selection, 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).

5.6 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, or PGA_CROSSOVER.UNIFORM for one-point, two-point, or uniform crossover, respectively. The default is two-point crossover. By default the crossover rate is 0.5. It may be set to 0.6 by PGASetCrossoverProb(ctx, 0.6), for example.

Uniform crossover is parameterized by \(p_u\), the probability of swapping two parent bit values [8]. By default, \(p_u = 0.5\). The function call PGASetUniformCrossoverProb(ctx, 0.7) will set \(p_u = 0.7\).

5.7 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 four 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 0.02, for example, with PGASetMutationRealValue(ctx, 0.02). 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, 0.5). 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.

Some of the integer- and real-valued mutation operators may generate allele values outside the initialization range of that gene. If this happens, by default, the allele value will be reset to the lower (upper) value of the initialization range for that gene. By setting PGASetMutationBoundedFlag(ctx, PGA_FALSE) the allele values will not be reset if they fall outside of the initialization range.

### 5.8 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 5.7 for mutating character strings.

### 5.9 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 5.1, 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

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

and in Fortran

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

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 6.) As shown below, $p$ and $pop$ are used for reading (and sometimes writing) allele values. Sample evaluation functions are shown in Figures 3.1 and 3.2, and online in the ./examples directory.

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. Therefore, the user's evaluation value is mapped to a nonnegative and monotonically increasing
fitness value. 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.

A linear rank fitness function \([2, 10]\) is given by

\[
Min + (Max - Min) \frac{\text{rank}(p) - 1}{N - 1},
\]

where \(\text{rank}(p)\) is the index of string \(p\) 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

\[
K - (\text{rank}(p) \ast C),
\]

where \(K\) and \(C\) are the constants \(\sigma \ast N\) and \(\sigma\), where \(\sigma\) 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_MAX) 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 PGASetFitnessMaxValue(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.

5.10 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

```c
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

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

If the data type is PGA_DATATYPE_INTEGER

```c
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

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

If the data type is PGA_DATATYPE_REAL

```c
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

```c
PGASetRealAllele(ctx, p, pop, i, 123.456);
```
If the data type is PGA_DATATYPE_CHARACTER

```c
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

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

### 5.10.1 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

\[
k = \sum_{i=1}^{N} b_i 2^{i-1},
\]

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

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

The function PGAGetIntegerFromBinary will encode an integer as a binary string. For example, to encode the integer 564 as a 12-bit binary string\(^2\) in the substring defined by bits 12–23, use

```c
PGAGetIntegerFromBinary(ctx, p, pop, 12, 23, 564);
```

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. (5.3) and may alternatively be used for representing integer (or real, see below) values in a binary string. The second and third columns in Table 5.1 show how the integers 0–7 are mapped to Eq. (5.3) 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

```c
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

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

### 5.10.2 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 5.10.1. 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

\(^2\)Even 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.

\(^3\)Technically, this is known as a Hamming cliff.
Table 5.1: Binary and Gray Codes

<table>
<thead>
<tr>
<th>k</th>
<th>Eq. (5.3)</th>
<th>Gray code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>100</td>
</tr>
</tbody>
</table>

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.

5.10.3 Example

As an example, suppose the user has a real-valued function f of three real variables x₁, x₂, and x₃. Further, the variables are constrained as follows.

\[-10 \leq x₁ \leq 0\]
\[0 \leq x₂ \leq 10\]
\[-10 \leq x₃ \leq 10\]

The user wishes to use 10 bits for the binary representation of x₁ and x₂, and 20 bits for the binary representation of x₃ (perhaps for higher accuracy), and a Gray code encoding. This may be done as follows.

```c
#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 $f$.

## 5.11 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.

## 5.12 Utility Functions

### 5.12.1 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)`. 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.

### 5.12.2 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.

### 5.12.3 Miscellaneous

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

`PGAUpdateOffline(ctx, pop)` (also `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

```c
PGAError(ctx, "popindex=-1", 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`.

27
5.13 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 12 for the function call equivalents.

- pgahelp
- pgahelp debug
- pgadbg <level>
- pgaddebug <level>
- pgaversion

get this message
list of debug options
set debug option
set debug option
Print current PGAPack version number, parallel or sequential, and debug or optimized
Chapter 6
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 10 shows how they may be executed in parallel.

6.1 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 \( \text{pop} \) is referred to by the 2-tuple \( (p, \text{pop}) \) and the value of gene \( i \) in that string by the 3-tuple \( (i, p, \text{pop}) \). In PGAPack, \( \text{pop} \) must be one of the two symbolic constants \( \text{PGA.JLDPOP} \) or \( \text{PGA.JEWPOP} \) 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 \( \text{PGA.TEMP1} \) or \( \text{PGA.TEMP2} \), to reference one of the two temporary locations, respectively.

6.2 Simple Sequential Example

The example in Figure 6.1 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 3.1. 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, referred to initially by the symbolic constant \( \text{PGA.JLDPOP} \), 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 6.1 and discussed earlier in Sections 5.1 and 5.9. The PGAFitness function maps the user’s evaluation function values into fitness values.

The while loop runs the genetic algorithm. PGAOne returns PGA_TRUE if any of the specified stopping criteria have been met, otherwise PGA_FALSE. PGASelect performs selection on population PGA.JLDPOP. 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);

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 6.1: 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 10) 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 10.2.

6.3 Complex Example

The primary difference between the "complex" example in Figure 6.2 and the "simple" example in Figure 6.1 is that the steps encapsulated by PGARunMutationAndCrossover have been written out explicitly. The function PGASortPop sorts a population according to the criteria specified by PGASetPopReplacementType (Section 5.2). 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 mi and m2 from population PGA_OLDPOP to create two child strings in the temporary locations PGA_TEMPI 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 6.1, 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);

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

    ctx = PGACreate(&argc, argv, PGA_DATATYPE_BINARY, 100, PGA_MAXIMIZE);
PAGASetUp(ctx);
    probcross = PGAGetCrossoverProb(ctx);
    popsize = PGAGetPopSize(ctx);
    numreplace = PGAGetNumReplaceValue(ctx);
    PGEvaluate(ctx, PGA_OLDPOP, evaluate, NULL);
PAGAFitness (ctx, PGA_OLDPOP);

    while(!PGADone(ctx, NULL)) {
        PGEvaluate(ctx, PGA_OLDPOP, evaluate, NULL);
PAGAFitness (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) {
            mi = PAGASelectNextIndex(ctx);
            m2 = PAGASelectNextIndex(ctx);
            if(PGARandomFlip(ctx, probcross)) {
                PGACrossover(ctx, mi, 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+i,PGA_NEWPOP);
                n += 2;
            } else {
                PGACopyIndividual (ctx, mi, PGA_OLDPOP, n, PGA_NEWPOP);
                PGACopyIndividual (ctx, m2, PGA_OLDPOP, n+i, PGA_NEWPOP);
                n += 2;
            }
        }
PGEvaluate(ctx, PGA_NEWPOP, evaluate, NULL);
PAGAFitness (ctx, PGA_NEWPOP);
PAGAPrintReport(ctx, stdout, PGA_NEWPOP);
PGAUpdateGeneration(ctx, NULL);
    }
    PGADestroy(ctx);
    return 0;
}

Figure 6.2: Example of Explicit Usage
6.4 Explicit PGAPack Functions

This section briefly discusses other functions not shown in the previous examples or discussed in Chapter 5. 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.

The restart operator described earlier in Section 5.8 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_OLDPOP, PGA_NEWPOP, 20) returns PGA-TRUE if string p in population PGA_OLDPOP is a duplicate of any of the first 20 strings in population PGA_NEWPOP. PGAChange(ctx, p, PGA_OLDPOP) repeatedly applies the mutation operator to string p in population PGA_OLDPOP 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 10 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_OLDPOP) returns the evaluation function value. PGAGetEvaluationUpToDateFlag(ctx, p, PGA_OLDPOP) 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_OLDPOP) 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_OLDPOP, 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_OLDPOP, PGA_FALSE) sets the status of the Boolean flag of string p in population PGA_OLDPOP 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. PGAIndex(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_OLDPOP) returns the index of the string in population PGA_OLDPOP with the worst evaluation function value. PGAGetBestIndex(ctx, PGA_OLDPOP) returns the index of the string in population PGA_OLDPOP with the best evaluation function value.
Chapter 7

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.

7.1 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, and a generic function called at the end of each GA iteration.

The function call `PGASetUserFunction(ctx, PGA_USERFUNCTION_MUTATION, mymute)` will cause PGAPack to execute the function `myme` whenever the mutation operator is called. Table 7.1 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 7.2. The files `./examples/templates/ufnative.c` and `./examples/templates/ufnative.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 5.3) 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` should only be called by the master process (see Chapter 10).

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

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Symbolic Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>PGA_USERFUNCTION_INITSTRING</td>
</tr>
<tr>
<td>Crossover</td>
<td>PGA_USERFUNCTION_CROSSOVER</td>
</tr>
<tr>
<td>Mutation</td>
<td>PGA_USERFUNCTION_MUTATION</td>
</tr>
<tr>
<td>Duplicate Checking</td>
<td>PGA_USERFUNCTION_DUPLICATE</td>
</tr>
<tr>
<td>String Printing</td>
<td>PGA_USERFUNCTION_PRINTSTRING</td>
</tr>
<tr>
<td>Termination Criteria</td>
<td>PGA_USERFUNCTION_STOPCOND</td>
</tr>
<tr>
<td>End of generation</td>
<td>PGA_USERFUNCTION_ENDOFGEN</td>
</tr>
</tbody>
</table>
before the population pointers (PGA.NEWPOP, PGA.OLDPOP) 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.

7.2 Example Problem: C

The example problem in Figure 7.1 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 PGASetUserFunction function 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 $[1, L]$.

7.3 Example Problem: Fortran

Figure 7.2 is the same example as in Figure 7.1 written in Fortran.
#include <pgapack.h>

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

int main( int argc, char **argv ) {
    PGAContext *ctx;
    int i, maxiter;
    ctx = PAGACreate ($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) {
    int stringlen, i, sum = 0;
    stringlen = PGAGetStringLength(ctx);
    for (i = 0; i < stringlen; i++)
        sum += PGAGetIntegerAllele(ctx, p, pop, i);
    return ((double)sum);
}

Figure 7.1: PAGApack 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)
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, i, 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 7.2: PGAPack Fortran Example Using Custom Mutation Operator
Chapter 8

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.

8.1 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 8.1, 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 8.1 are given in Table 8.2. The file ./examples/templates/uf_new.c contains template routines for these functions.

While PGAPack requires that the user supply all the functions in Table 8.1, 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.

8.2 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 8.1 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>
<thead>
<tr>
<th>Functionality</th>
<th>Symbolic Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory allocation</td>
<td>PGA_USERFUNCTION_CREATESTRING</td>
</tr>
<tr>
<td>String packing</td>
<td>PGA_USERFUNCTION_BUILDDATATYPE</td>
</tr>
<tr>
<td>Mutation</td>
<td>PGA_USERFUNCTION_MUTATION</td>
</tr>
<tr>
<td>Crossover</td>
<td>PGA_USERFUNCTION_CROSSOVER</td>
</tr>
<tr>
<td>String printing</td>
<td>PGA_USERFUNCTION_PRINTSTRING</td>
</tr>
<tr>
<td>String copying</td>
<td>PGA_USERFUNCTION_COPYSTRING</td>
</tr>
<tr>
<td>Duplicate checking</td>
<td>PGA_USERFUNCTION_DUPLICATE</td>
</tr>
</tbody>
</table>
Table 8.2: Calling Sequences for New Data Type Functions

<table>
<thead>
<tr>
<th>Symbolic Constant</th>
<th>Return</th>
<th>Function Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGA_USERFUNCTION_CREATESTRING</td>
<td>void</td>
<td>(PGAContext*, int, int, int, int)</td>
</tr>
<tr>
<td>PGA_USERFUNCTION_BUILDDATATYPE</td>
<td>int</td>
<td>(PGAContext*, int, int)</td>
</tr>
<tr>
<td>PGA_USERFUNCTION_MUTATION</td>
<td>int</td>
<td>(PGAContext*, int, int, double)</td>
</tr>
<tr>
<td>PGA_USERFUNCTION_CROSSOVER</td>
<td>void</td>
<td>(PGAContext*, int, int, int, int, int, int)</td>
</tr>
<tr>
<td>PGA_USERFUNCTION_PRINTSTRING</td>
<td>void</td>
<td>(PGAContext*, FILE *, int, int)</td>
</tr>
<tr>
<td>PGA_USERFUNCTIONCOPYSTRING</td>
<td>int</td>
<td>(PGAContext*, int, int, int, int)</td>
</tr>
<tr>
<td>PGA_USERFUNCTION_DUPLICATE</td>
<td>int</td>
<td>(PGAContext*, int, int, int, int)</td>
</tr>
</tbody>
</table>

#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 DuplicateString (PGAContext *, int, 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);
    PGRun (ctx, Evaluate);
    PGADestroy (ctx);
    return (0);
}

Figure 8.1: Main Program for Structure Data Type
void CreateString(PGAC0ntext *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'\n);
        exit(1);
    }
    ligand_ptr = (ligand *)new->chrom;
    if (InitFlag) {
        for (i = 0; i < 3; i++)
            ligand_ptr->t[i] = PGARandomOl(ctx, 0) * 20.0 - 10.0;
        for (i = 3; i < 6; i++)
            ligand_ptr->t[i] = PGARandomOl(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 8.2: 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 8.2 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.OLDPOP (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 5.4.)

Figure 8.3 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 8.4 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;
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 8.3: Mutation for Structure Data Type

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

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

Figure 8.6 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 8.7 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 8.4: 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 8.5: Duplicate Testing for Structure Data Type
double Evaluate(PGAContext *ctx, int p, int pop) {
    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 8.6: 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_PGAIndividual;
    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 on 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_PGAIndividual);
    MPI_Type_commit(&DT_PGAIndividual);
    return (DT_PGAIndividual);
}

Figure 8.7: Message Packing Function for Structure Data Type
Chapter 9

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.

One way to run a hybrid GA and use PGARun is to use the PGASetUserFunction discussed in Chapter 7 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 9.1 is a version of the Maxbit problem given in Section 3.1. 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);
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);
        hillclimb (ctx, PGA_OLDPOP);
        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, strlen;
    popsize = PGAGetPopSize(ctx);
    strlen = PGAGetStringLength(ctx);
    for (p=0; p<popsize; p++) {
        i = PGARandomInterval(ctx, 0, strlen-1);
        PGASetBinaryAllele (ctx, p, pop, i, 1);
        PGASetEvaluationUpToDateFlag (ctx, p, pop, PGA_FALSE);
    }
}

Figure 9.1: Hill-Climbing Heuristic for Maxbit Example

45
Chapter 10

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 (see Section 2.4).

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 processes\(^1\).

10.1 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 3 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.

10.2 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, PGAReceiveIndividual, 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.

\(^1\)In 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.

`PGAReceiveIndividual` is the complementary function to `PGASendIndividual`. For example, `PGAReceiveIndividual (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 `PGAReceiveIndividual`. 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`, `PGAReceiveIndividual`, and `PGASendReceiveIndividual` send and receive. The result is of type `MPI.Datatype`.

### 10.3 Example

Figure 10.1 is a parallel version of the example in Figure 6.1. 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.

### 10.4 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);

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 10.1: Simple Parallel Example of Explicit Usage
Chapter 11

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 5.13).
- Fortran allows custom usage with native data types (Chapter 7), but not with new data types (Chapter 8).
- 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 differences\(^1\). 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

  ```fortran
  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.

\(^1\) More generally, these issues arise whenever the size of a Fortran integer is less than the size of a pointer.
Chapter 12
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. See Section 2.4.

The trace facility uses the concept of a debug level. For example, executing the Maxbit example (Figure 3.1) with a debug level of 12, maxbit -pgadbg 12, will print the output shown in Figure 12.1. 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 12.1. 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 -pgadbg <debug_level> or -pgaddebug <debug_level>. Several forms of the <debug_level> argument are allowed. -pgadbg 12 will trace entering all high-level functions as shown in Figure 12.1. -pgadbg 12,13 or -pgadbg 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 12.1. 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 12.1: 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>
<thead>
<tr>
<th>Debug Level (decimal)</th>
<th>Debug Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Trace all debug prints</td>
</tr>
<tr>
<td>11</td>
<td>Trace high-level functions</td>
</tr>
<tr>
<td>12</td>
<td>Trace all function entries</td>
</tr>
<tr>
<td>13</td>
<td>Trace all function exits</td>
</tr>
<tr>
<td>20</td>
<td>Trace high-level parallel functions</td>
</tr>
<tr>
<td>21</td>
<td>Trace all parallel functions</td>
</tr>
<tr>
<td>22</td>
<td>Trace all send calls</td>
</tr>
<tr>
<td>23</td>
<td>Trace all receive calls</td>
</tr>
<tr>
<td>30</td>
<td>Trace Binary functions</td>
</tr>
<tr>
<td>32</td>
<td>Trace Integer functions</td>
</tr>
<tr>
<td>34</td>
<td>Trace Real functions</td>
</tr>
<tr>
<td>36</td>
<td>Trace Character functions</td>
</tr>
<tr>
<td>40</td>
<td>Trace population creation functions</td>
</tr>
<tr>
<td>42</td>
<td>Trace select functions</td>
</tr>
<tr>
<td>44</td>
<td>Trace mutation functions</td>
</tr>
<tr>
<td>46</td>
<td>Trace crossover functions</td>
</tr>
<tr>
<td>48</td>
<td>Trace function evaluation functions</td>
</tr>
<tr>
<td>50</td>
<td>Trace fitness calculation functions</td>
</tr>
<tr>
<td>52</td>
<td>Trace duplicate checking functions</td>
</tr>
<tr>
<td>54</td>
<td>Trace restart functions</td>
</tr>
<tr>
<td>56</td>
<td>Trace reporting functions</td>
</tr>
<tr>
<td>58</td>
<td>Trace stopping functions</td>
</tr>
<tr>
<td>60</td>
<td>Trace sorting functions</td>
</tr>
<tr>
<td>62</td>
<td>Trace random number functions</td>
</tr>
<tr>
<td>64</td>
<td>Trace system routines</td>
</tr>
<tr>
<td>66</td>
<td>Trace utility functions</td>
</tr>
<tr>
<td>80</td>
<td>Trace memory allocations</td>
</tr>
<tr>
<td>82</td>
<td>Trace variable print statements</td>
</tr>
</tbody>
</table>
Part III

Appendixes
Appendix A

Default Values
<table>
<thead>
<tr>
<th>CONCEPT</th>
<th>DEFAULT</th>
<th>SET WITH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
<td>PGASetPopSize</td>
</tr>
<tr>
<td>Copied for population replacement</td>
<td>PGA_POPREPL_BEST</td>
<td>PGASetPopReplacementType</td>
</tr>
<tr>
<td>Stopping rule</td>
<td>PGA_STOP_MAXITER</td>
<td>PGASetStoppingRuleType</td>
</tr>
<tr>
<td>Maximum iterations</td>
<td>1000</td>
<td>PGASetMaxGAIterValue</td>
</tr>
<tr>
<td>Maximum no change iters</td>
<td>100</td>
<td>PGASetMaxNoChangeValue</td>
</tr>
<tr>
<td>Max. population homogeneity before stopping</td>
<td>95</td>
<td>PGASetMaxSimilarityValue</td>
</tr>
<tr>
<td>Number of new strings to generate</td>
<td>10</td>
<td>PGASetNumReplaceValue</td>
</tr>
<tr>
<td>Apply mutation and crossover</td>
<td>PGA_FALSE</td>
<td>PGASetMutationAndCrossoverFlag</td>
</tr>
<tr>
<td>Apply mutation or crossover</td>
<td>PGA_TRUE</td>
<td>PGASetMutationOrCrossoverFlag</td>
</tr>
<tr>
<td>Crossover type</td>
<td>PGA_CROSSOVER_TWOPT</td>
<td>PGASetCrossoverType</td>
</tr>
<tr>
<td>Probability of crossover</td>
<td>0.85</td>
<td>PGASetCrossoverProb</td>
</tr>
<tr>
<td>Uniform crossover bias</td>
<td>0.6</td>
<td>PGASetUniformCrossoverProb</td>
</tr>
<tr>
<td>Mutation type (Real strings)</td>
<td>PGA_MUTATION_GAUSSIAN</td>
<td>PGASetMutationType</td>
</tr>
<tr>
<td>Mutation type (Integer strings)</td>
<td>PGA_MUTATION_PERMUTE</td>
<td>PGASetMutationType</td>
</tr>
<tr>
<td>Mutation type (Character strings)</td>
<td>Same as initialization</td>
<td>PGASetCharacterInitType</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>1/L</td>
<td>PGASetMutationProb</td>
</tr>
<tr>
<td>Real mutation constant</td>
<td>0.1</td>
<td>PGASetMutationRealValue</td>
</tr>
<tr>
<td>Integer mutation constant</td>
<td>1</td>
<td>PGASetMutationIntegerValue</td>
</tr>
<tr>
<td>Mutation range bounded</td>
<td>PGA_TRUE</td>
<td>PGASetMutationBoundedFlag</td>
</tr>
<tr>
<td>Select type</td>
<td>PGA_SELECT_TOURNAMENT</td>
<td>PGASetSelectType</td>
</tr>
<tr>
<td>Probabilistic binary tournament parameter</td>
<td>0.6</td>
<td>PGASetPTournamentProb</td>
</tr>
<tr>
<td>Use restart operator</td>
<td>PGA_FALSE</td>
<td>PGASetRestartFlag</td>
</tr>
<tr>
<td>Restart frequency</td>
<td>50</td>
<td>PGASetRestartFrequencyValue</td>
</tr>
<tr>
<td>Restart allele mutation rate</td>
<td>0.5</td>
<td>PGASetRestartAlleleChangeProb</td>
</tr>
<tr>
<td>Allow duplicate strings</td>
<td>PGA_FALSE</td>
<td>PGASetNoDuplicatesFlag</td>
</tr>
<tr>
<td>Fitness type</td>
<td>PGA_FITNESS_RAW</td>
<td>PGASetFitnessType</td>
</tr>
<tr>
<td>Fitness type for minimization</td>
<td>PGA_FITNESSMIN_CMAX</td>
<td>PGASetFitnessMinType</td>
</tr>
<tr>
<td>Multiplier for minimization problems</td>
<td>1.01</td>
<td>PGASetCMaxValue</td>
</tr>
<tr>
<td>Parameter MAX in fitness by ranking</td>
<td>1.2</td>
<td>PGASetMaxFitnessRank</td>
</tr>
<tr>
<td>Frequency of statistics printing</td>
<td>10</td>
<td>PGASetPrintFrequencyValue</td>
</tr>
<tr>
<td>Print strings</td>
<td>PGA_FALSE</td>
<td>PGASetPrintOptions</td>
</tr>
<tr>
<td>Print offline statistics</td>
<td>PGA_FALSE</td>
<td>PGASetPrintOptions</td>
</tr>
<tr>
<td>Print online statistics</td>
<td>PGA_FALSE</td>
<td>PGASetPrintOptions</td>
</tr>
<tr>
<td>Print best string</td>
<td>PGA_FALSE</td>
<td>PGASetPrintOptions</td>
</tr>
<tr>
<td>Print worst string</td>
<td>PGA_FALSE</td>
<td>PGASetPrintOptions</td>
</tr>
<tr>
<td>Print Hamming distance</td>
<td>PGA_FALSE</td>
<td>PGASetPrintOptions</td>
</tr>
<tr>
<td>Randomly initialize population</td>
<td>PGA_TRUE</td>
<td>PGASetRandomInitFlag</td>
</tr>
<tr>
<td>Probability of initializing a bit to one</td>
<td>0.5</td>
<td>PGASetBinaryInitProb</td>
</tr>
<tr>
<td>How to initialize real strings</td>
<td>Range</td>
<td>PGASetRealInitRange</td>
</tr>
<tr>
<td>Real initialization range</td>
<td>[0, 1]</td>
<td>PGASetRealInitRange</td>
</tr>
<tr>
<td>How to initialize integer strings</td>
<td>Permutation</td>
<td>PGASetIntegerInitPermute</td>
</tr>
<tr>
<td>Integer initialization range</td>
<td>[0, (L-1)]</td>
<td>PGASetIntegerInitPermute</td>
</tr>
<tr>
<td>How to initialize character strings</td>
<td>PGA_CINIT_LOWER</td>
<td>PGASetCharacterInitFlag</td>
</tr>
<tr>
<td>Seed random number with clock</td>
<td>PGA_TRUE</td>
<td>PGASetRandomSeed</td>
</tr>
<tr>
<td>Default MPI communicator</td>
<td>MPI_COMM_WORLD</td>
<td>PGASetCommunicator</td>
</tr>
</tbody>
</table>

\(L\) is the string length.
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
  - PGADATATYPE_BINARY
  - PGADATATYPE_INTEGER
  - PGADATATYPE_REAL
  - PGADATATYPE_CHARACTER
  - PGADATATYPE_USER

- String Types
  - PGABinary
  - PGAInteger
  - PGAReal
  - PGACharacter

- 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_PRINTFVAR
  – 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_RECIPECAL
  – 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.Morep
  - PGA.PopRepl.Random.Rep

• Initialization Options
  - PGA.CInit.Lower
  - PGA.CInit.Upper
  - PGA.CInit.Mixed
  - PGA.Init.Permute
  - PGA.Init.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

• 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
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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Datatype</td>
<td>PGABuildDatatype(PGAC0ntext *ctx, int p, int pop)</td>
</tr>
<tr>
<td>void</td>
<td>PGACConvert(PGAC0ntext *ctx, int p, int pop)</td>
</tr>
<tr>
<td>int</td>
<td>PGACheckStoppingConditions(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGACheckSum(PGAC0ntext *ctx, int p, int pop)</td>
</tr>
<tr>
<td>void</td>
<td>PGAClearDebugLevel(PGAC0ntext *ctx, int level)</td>
</tr>
<tr>
<td>void</td>
<td>PGAClearDebugLevelByDate(PGAC0ntext *ctx, char *funcname)</td>
</tr>
<tr>
<td>void</td>
<td>PGACopyIndividual(PGAC0ntext *ctx, int p, int pop1, int p2, int pop2)</td>
</tr>
<tr>
<td>PGAC0ntext*</td>
<td>PGACreate(int *argv, char **argv, int datatype, int len, int maxormin)</td>
</tr>
<tr>
<td>void</td>
<td>PGACr0ss(PGAC0ntext *ctx, int p, int pop, int c1, int c2, int pop2)</td>
</tr>
<tr>
<td>void</td>
<td>PGADebugPrint(PGAC0ntext *ctx, int level, char *funcname, char *msg, int datatype, void *data)</td>
</tr>
<tr>
<td>void</td>
<td>PGADestroy(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGADone(PGAC0ntext *ctx, MPI_Comm comm)</td>
</tr>
<tr>
<td>int</td>
<td>PGADuplicate(PGAC0ntext *ctx, int p, int pop1, int pop2, int n)</td>
</tr>
<tr>
<td>void</td>
<td>PGACencodeIntegerAsBinary(PGAC0ntext *ctx, int p, int pop, int start, int end, int val)</td>
</tr>
<tr>
<td>void</td>
<td>PGACencodeIntegerAsGrayCode(PGAC0ntext *ctx, int p, int pop, int start, int end, int val)</td>
</tr>
<tr>
<td>void</td>
<td>PGACencodeRealAsBinary(PGAC0ntext *ctx, int p, int pop, int start, int end, double low, double high, double val)</td>
</tr>
<tr>
<td>void</td>
<td>PGACencodeRealAsGrayCode(PGAC0ntext *ctx, int p, int pop, int start, int end, double low, double high, double val)</td>
</tr>
<tr>
<td>void</td>
<td>PGAEval0uate(PGAC0ntext *ctx, char *msg, int level, int datatype, void *data)</td>
</tr>
<tr>
<td>void</td>
<td>PFAFitness(PGAC0ntext *ctx, int popindex)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetBestIndex(PGAC0ntext *ctx, int pop)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetBinaryAllele(PGAC0ntext *ctx, int p, int pop, int i)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetBinaryInitProb(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>char</td>
<td>PGAGetCharacterAllele(PGAC0ntext *ctx, int p, int pop, int i)</td>
</tr>
<tr>
<td>MPI_Comm</td>
<td>PGAGetCommunicator(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetCrossoverProb(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetCrossoverType(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetAddress(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetAddress(PGAC0ntext *ctx, int p, int pop)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetAddressUpToDateFlag(PGAC0ntext *ctx, int p, int pop)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetFitness(PGAC0ntext *ctx, int p, int pop)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetFitnessCmaxValue(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetFitnessMinType(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetFitnessType(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetGAlterValue(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetIntegerAllele(PGAC0ntext *ctx, int p, int pop, int i)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetIntegerFromBinary(PGAC0ntext *ctx, int p, int pop, int start, int end)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetIntegerFromGrayCode(PGAC0ntext *ctx, int p, int pop, int start, int end)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetIntegerInitType(PGAC0ntext *ctx)</td>
</tr>
<tr>
<td>Type</td>
<td>Function</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------------</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetMaxFitnessRank(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMaxGAlterValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMaxIntegerInitValue(PGAContext *ctx, int i)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetMaxMachineDoubleValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMaxMachineIntValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetMaxRealInitValue(PGAContext *ctx, int i)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMinIntegerInitValue(PGAContext *ctx, int i)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetMinMachineDoubleValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMinMachineIntValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetMinRealInitValue(PGAContext *ctx, int i)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMutationAndCrossoverFlag(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMutationBoundedFlag(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMutationIntegerValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMutationOrCrossoverFlag(PGAContext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetMutationProb(PGAContext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetMutationRealValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetMutationType(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetNoDuplicatesFlag(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetNumProcs(PGAContext *ctx, MPI_Comm comm)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetNumReplaceValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetOptDirFlag(PGAContext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetPTournamentProb(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetPopReplaceType(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetPopSize(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetPrintFrequencyValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetRandomInitFlag(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetRandomSeed(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetRank(PGAContext *ctx, MPI_Comm comm)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetRealAllele(PGAContext *ctx, int p, int pop, int i)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetRealFromBinary(PGAContext *ctx, int p, int pop, int start, int end, double lower, double upper)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetRealInitType(PGAContext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetRestartAlleleChangeProb(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetRestartFlag(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetRestartFrequencyValue(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetSelectType(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetSortedPopIndex(PGAContext *ctx, int n)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetStoppingRuleType(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetStringLength(PGAContext *ctx)</td>
</tr>
<tr>
<td>double</td>
<td>PGAGetUniformCrossoverProb(PGAContext *ctx)</td>
</tr>
<tr>
<td>int</td>
<td>PGAGetWorstIndex(PGAContext *ctx, int pop)</td>
</tr>
<tr>
<td>double</td>
<td>PGAHammingDistance(PGAContext *ctx, int popindex)</td>
</tr>
<tr>
<td>double</td>
<td>PGAMean(PGAContext *ctx, double *a, int n)</td>
</tr>
<tr>
<td>int</td>
<td>PGAMutate(PGAContext *ctx, int p, int pop)</td>
</tr>
<tr>
<td>void</td>
<td>PGAPrintContextVariable(PGAContext *ctx, FILE *fp)</td>
</tr>
<tr>
<td>void</td>
<td>PGAPrintIndividual(PGAContext *ctx, FILE *fp, int p, int pop)</td>
</tr>
<tr>
<td>void</td>
<td>PGAPrintPopulation(PGAContext *ctx, FILE *fp, int pop)</td>
</tr>
<tr>
<td>void</td>
<td>PGAPrintReport(PGAContext *ctx, FILE *fp, int pop)</td>
</tr>
<tr>
<td>void</td>
<td>PGAPrintString(PGAContext *ctx, FILE *file, int p, int pop)</td>
</tr>
<tr>
<td>void</td>
<td>PGAPrintVersionNumber(PGAContext *ctx)</td>
</tr>
<tr>
<td>Type</td>
<td>Function</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>double</td>
<td>PGARandom01(PGAContext *ctx, int newseed)</td>
</tr>
<tr>
<td>int</td>
<td>PGARandomFlip(PGAContext *ctx, double p)</td>
</tr>
<tr>
<td>double</td>
<td>PGARandomGaussian(PGAContext *ctx, double mean, double sigma)</td>
</tr>
<tr>
<td>int</td>
<td>PGARandomInterval(PGAContext *ctx, int start, int end)</td>
</tr>
<tr>
<td>double</td>
<td>PGARandomUniform(PGAContext *ctx, double start, double end)</td>
</tr>
<tr>
<td>int</td>
<td>PGARank(PGAContext *ctx, int p, int *order, int n)</td>
</tr>
<tr>
<td>void</td>
<td>PGAReceiveIndividual(PGAContext *ctx, int p, int pop, int source, int tag, MPLComm comm, MPLStatus *status)</td>
</tr>
<tr>
<td>void</td>
<td>PGARestart(PGAContext *ctx, int source_pop, int dest_pop)</td>
</tr>
<tr>
<td>int</td>
<td>PGARound(PGAContext *ctx, double x)</td>
</tr>
<tr>
<td>void</td>
<td>PGARun(PGAContext *ctx, double(*evaluate)(PGAContext *c, int p, int pop))</td>
</tr>
<tr>
<td>void</td>
<td>PGARunGM(PGAContext *ctx, double(*f)(PGAContext *, int, int), MPLComm comm)</td>
</tr>
<tr>
<td>void</td>
<td>PGARunMutationAndCrossover(PGAContext *ctx, int oldpop, int newpop)</td>
</tr>
<tr>
<td>void</td>
<td>PGARunMutationOrCrossover(PGAContext *ctx, int oldpop, int newpop)</td>
</tr>
<tr>
<td>void</td>
<td>PGASelect(PGAContext *ctx, int popix)</td>
</tr>
<tr>
<td>int</td>
<td>PGASelectNextIndex(PGAContext *ctx)</td>
</tr>
<tr>
<td>void</td>
<td>PGASendIndividual(PGAContext *ctx, int p, int pop, int dest, int tag, MPLComm comm)</td>
</tr>
<tr>
<td>void</td>
<td>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, MPLComm comm, MPLStatus *status)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetBinaryAllele(PGAContext *ctx, int p, int pop, int i, int val)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetBinaryInitProb(PGAContext *ctx, double probability)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetCharacterAllele(PGAContext *ctx, int p, int pop, int i, char value)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetCharacterInitType(PGAContext *ctx, int value)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetCommunicator(PGAContext *ctx, MPLComm comm)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetCrossoverProb(PGAContext *ctx, double crossover_prob)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetCrossoverType(PGAContext *ctx, int crossover_type)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetDebugLevel(PGAContext *ctx, int level)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetDebugLevelByName(PGAContext *ctx, char *funname)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetEvaluation(PGAContext *ctx, int p, int pop, double val)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetEvaluationUpToDateFlag(PGAContext *ctx, int p, int pop, int status)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetFitnessCmaxValue(PGAContext *ctx, double val)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetFitnessMinType(PGAContext *ctx, int fitness_type)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetFitnessType(PGAContext *ctx, int fitness_type)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetIntegerAllele(PGAContext *ctx, int p, int pop, int i, int value)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetIntegerInitPermute(PGAContext *ctx, int min, int max)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetIntegerInitRange(PGAContext *ctx, int *min, int *max)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMaxFitnessRank(PGAContext *ctx, double fitness_rank_max)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMaxGAIterValue(PGAContext *ctx, int maxiter)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMaxNoChangeValue(PGAContext *ctx, int max_no_change)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMaxSimilarityValue(PGAContext *ctx, int max_similarity)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMutationAndCrossoverFlag(PGAContext *ctx, int flag)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMutationBoundedFlag(PGAContext *ctx, int flag)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMutationIntegerValue(PGAContext *ctx, int val)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMutationOrCrossoverFlag(PGAContext *ctx, int flag)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMutationProb(PGAContext *ctx, double mutation_prob)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMutationRealValue(PGAContext *ctx, double val)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetMutationType(PGAContext *ctx, int mutation_type)</td>
</tr>
<tr>
<td>Type</td>
<td>Function</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>void</td>
<td>PGASetNoDuplicatesFlag(PGAContext *ctx, int no_dup)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetNumReplaceValue(PGAContext *ctx, int pop_replace)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetPTournamentProb(PGAContext *ctx, double ptournament_prob)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetPopReplaceType(PGAContext *ctx, int pop_replace)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetPopSize(PGAContext *ctx, int popsize)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetPrintFrequencyValue(PGAContext *ctx, int print_freq)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetPrintOptions(PGAContext *ctx, int option)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetRandomInitFlag(PGAContext *ctx, int RandomBoolean)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetRandomSeed(PGAContext *ctx, int seed)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetRealAllele(PGAContext *ctx, int p, int pop, int i, double value)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetRealInitPercent(PGAContext *ctx, double *median, double *percent)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetRealInitRange(PGAContext *ctx, double *min, double *max)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetRestartAlleleChangeProb(PGAContext *ctx, double prob)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetRestartFlag(PGAContext *ctx, int Val)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetRestartFrequencyValue(PGAContext *ctx, int numiter)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetSelectType(PGAContext *ctx, int select_type)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetStoppingRuleType(PGAContext *ctx, int stoprule)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetUniformCrossoverProb(PGAContext *ctx, double uniform_cross_prob)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetUp(PGAContext *ctx)</td>
</tr>
<tr>
<td>void</td>
<td>PGASetUserFunction(PGAContext *ctx, int constant, void *f)</td>
</tr>
<tr>
<td>double</td>
<td>PGAStddev(PGAContext *ctx, double *a, int n, double mean)</td>
</tr>
<tr>
<td>void</td>
<td>PGAUpdateGeneration(PGAContext *ctx, MPI_Comm comm)</td>
</tr>
<tr>
<td>void</td>
<td>PGAUsage(PGAContext *ctx)</td>
</tr>
</tbody>
</table>

**Fortran 77 Bindings**

Use the rules defined in Chapter 11 (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 [4]. 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 distributes 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, \ldots, \( n - 1 \). 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 [5, 6].

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:

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

and

```c
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:

```c
#include "pgapack.h"

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

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) {
        /* Process 0 has a dialog */
        printf("String length? ");
        /* with the user and */
        scanf("%d", &len);
        /* broadcasts the user's */
    }
    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 11.

We used MPICH with the ch.p4 device and 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 -npe 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_EOLIB us
setenv MP_INFOLEVEL 0
setenv MP_HOLD_STDIN YES
setenv MP_PULSE 0
a.out
```

69
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 11.
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!

71
We have also seen the following error from not including `bmpif.h` in the main program:

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

- If the `ch_p4` device in MPICH is used to run 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
  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`.
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.
Bibliography


Distribution for ANL-95/18

Internal:

J. M. Beumer (10)
F. Y. Fradin
W. D. Gropp
D. M. Levine (10)
L. C. McInnes
G. W. Pieper
B. F. Smith
F. J. Stevens
R. L. Stevens
C. L. Wilkinson
TIS File

External:

DOE-OSTI, for distribution per UC-405 (52)
ANL-E Library
ANL-W Library
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
   F. Berman, University of California at LaJolla
   G. Cybenko, Dartmouth College
   T. DuPont, The University of Chicago
   J. G. Glimm, State University of New York at Stony Brook
   M. T. Heath, University of Illinois, Urbana
   E. F. Infante, University of Minnesota
   K. Kunen, University of Wisconsin at Madison
   R. E. O'Malley, University of Washington
   L. R. Petzold, University of Minnesota
   F. Howes, Dept. of Energy - Office of Computational and Technology Research
   D. Nelson, Dept. of Energy - Office of Computational and Technology Research