# The ANSI C Implementation of the Cowichan Problems

Gregory V. Wilson

Version 1.0

**Abstract**

This document describes the implementation of the Cowichan Problems, a small suite of programs used to assess and compare parallel programming systems.

## 1 Introduction

The Cowichan Problems are a set of 14 small applications designed to be used in assessing and comparing parallel programming systems. This document describes the ANSI C implementation of these programs, which serves as a reference implementation. It also describes two multi-threaded implementations: one uses a repeated fork-and-join model, while the other creates a fixed number of threads at the start of the program, and then repeatedly synchronizes them using barriers. Throughout this document, the purely sequential implementation is referred to as the "serial" implementation, while the other two are referred to as the "fork-join" and "barrier" implementations respectively.

Table 1 gives the name of each application, its single-character code, and a one-line description of it. For full descriptions of these problems, see the full paper which accompanies this documentation.

## 2 Organization

In order to simplify automatic re-compilation and execution of these programs, the code for each implementation is contained in a fixed number of files, with fixed names, in fixed sub-directories. For an implementation `z`, these directories and files are organized as shown in Table 2.

| Code | Name | Description |
|---|---|---|
| C | `chained` | chained version of all problems |
| E | `elastic` | elastic net approximation to the TSP |
| G | `gauss` | Gaussian elimination (without pivoting) |
| H | `half` | halving shuffle |
| I | `invperc` | invasion percolation |
| L | `life` | simulation of the Game of Life |
| M | `mandel` | Mandelbrot Set generation |
| N | `norm` | co-ordinate normalization |
| O | `outer` | vector-vector outer product |
| P | `product` | matrix-vector product |
| R | `randmat` | random matrix generation |
| S | `sor` | successive over-relaxation |
| T | `thresh` | histogram-based image thresholding |
| V | `vecdiff` | vector difference |
| W | `winnow` | masked point winnowing |

Table 1: Application Descriptions

The directory `app` is the most important. It holds the implementations of the individual applications, one per file, plus another file called `app.c`, which contains utility routines for doing such things as finding the minimum or maximum value in a matrix, copying one vector into another, and so on. In some implementations, one or more of the application implementation "files" may actually be a link to the file containing standard sequential implementation. At present, for example, both the fork-join and barrier implementation use the sequential implementation of the `invperc` application in this way.

A separate directory, `main`, holds the main drivers for the individual applications. These drivers provide the top-level procedure `main`, parse command-line arguments, and do file input and output where appropriate. Drivers are separated from implementations because the chained version of the problem set incorporates all of the individual applications under the control of a single driver. In practice, some or all of these may be links to the standard sequential drivers; in fact, whether or not this can be done is one sign of how well a particular parallel programming system modularizes parallelism.

A third directory, `hdr`, holds the definitions used in each version. The file `hdr/specific.h` is the only one included by the source code in the `app`

`z/app`: source for individual applications
- `app.c`: version-specific support routines
- `elastic.c`: source for elastic net application
- ...
- `winnow.c`: source for winnow application

`z/bin`: executables created during compilation
- `chain`: chained version of all applications
- `elastic`: stand-alone elastic net
- ...
- `winnow`: stand-alone winnowing

`z/hdr`: header files
- `generic.h`: generic definitions
- `gfx.h`: graphics interface definitions
- `proto.h`: function prototypes for individual applications
- `specific.h`: version-specific definitions
- `type.h`: type and constant definitions

`z/lib`: libraries
- `libapp.a`: support routines and individual applications

`z/main`: top-level drivers
- `chain.c`: driver for chained version of all applications
- `elastic.c`: driver for stand-alone elastic net
- ...
- `winnow.c`: driver for stand-alone point winnowing

`z/test`:
- *see Section 9.2*

Table 2: Code Organization

and `main` directories; it references all other header files which programs require. The `make` control file ensures that the compiler looks in the right place to resolve these references (Section 9.1). The other four file in this directory are: `generic.h`, which contains version-independent definitions of various constants; `gfx.h`, which contains function prototypes for the graphics interface; `proto.h`, containing function prototypes for the application implementations themselves; and `type.h`, which contains definitions of the data types used in the version. Definitions are split up in this way for the following reasons:

- Some programming systems of interest require special type definitions to indicate parallel structures (e.g. the `shape` definition in C*). Since an overriding design consideration was that no piece of code should more than once, type definitions had to be separated from other definitions.
- The interface to a particular application may require extra parameters, depending on the type of parallelism used. The barrier version described in this document, for example, requires a thread ID to be passed as an argument to every parallelized routine. Application function prototypes are therefore kept in a separate file `proto.h` so that they can be changed without affecting other definitions.
- The graphics utilities may eventually be dependent on the style of parallelism used, although they are not at present. A separate prototype file `gfx.h` was therefore created.

In the versions described in this document, `generic.h`, `gfx.h`, and `type.h` are all links to the equivalent standard sequential files. Only `proto.h` and `specific.h` reflect the idiosyncracies of particular parallel implementations.

# 3 Data Structures and Associated Definitions

Four basic data types are used in these applications: Booleans (`logical` variables to Fortran programmers), integers, real numbers, and weighted points consisting of real-valued $(x, y)$ coordinates and a non-negative integer weight $w$. Booleans are distinguished from integers, through a `typedef` of the name `bool`, for the benefit of systems which support compact implementation of Booleans (such as C*). Real number are created using `typedef` to define the name `real`. At present, all real values are `double`, but this may be changed in other programming systems. Finally, a structure is defined to represent individual points using:

```
typedef struct {
  real  x, y;        /* location */
  int   w;           /* weight */
} pt;
```

With these four primitive types in place, vectors and arrays of them are defined. These definitions depend upon the compile-time constant `MAXEXT`, which is the maximum extent of any structure along a single axis. (For a discussion of the "slushy" data model this embodies, see the accompanying paper.) The suffix `1D` is used to indicate a vector of size `MAXEXT` of a particular type; thus, `int1D` is `typedef`'d to be a vector of integers, `pt1D` a vector of points, and so on. The suffix `2D` is used to indicate a square array of this size, so that `real2D` is equivalent to `real[MAXEXT][MAXEXT]`. Finally, the suffix `1DX` is used to indicate an "extended" vector, which contains as many elements as a square array (i.e. $\texttt{MAXEXT}^2$). Temporary structures of this shape and size are needed in the `winnow` application.

Two enumerated types are defined as part of this suite as well. The first is `gfxCtrl_e`, and its values determine how the graphics interface behaves. The second, `dataDist_e`, is more important for present purposes. Its values enumerate the data and work distribution strategies supported in this application suite. Currently, only single-axis block and cyclic distributions are available, but others may be added in future implementations.

Finally, codes for individual applications, and miscellaneous parameters such as the maximum number of iterations to perform at a single point when generating the Mandelbrot Set, are defined in `generic.h`. These should be self-explanatory.

# 4 Utilities

The organization of the directory `generic` is slightly different from that of the other directories. Like them, it contains `app` and `main` directories holding application implementations and main drivers, and a `hdr` directory containing various definition files. The files in `app` and `main` are not used directly, however; instead, links to them are made from the directories containing other versions when the standard sequential implementation of an application or driver is to be recycled. For example, the serial, fork-join, and barrier versions all use the same code for `invperc`, and so all contain links to the generic version. Similarly, the main drivers in the fork-join version are identical with those used in the serial version, so both are implemented

using links to the generic version.

The `hdr` directory contains one non-standard file, called `util.h`. This is referenced by the utility code modules contained in the `util` directory. These modules implement routines which are used in all versions of this problem suite for such things as command-line argument parsing and error handling. The files in this directory are:

`arg.c`: command-line argument handling
`gfx.c`: standard graphics interface
`io.c`: file input and output
`misc.c`: miscellaneous (e.g. error handling)
`sch.c`: scheduling
`str.c`: string conversion

Of these, the graphics, I/O, and scheduling modules deserve further explanation.

## 4.1 Graphics

The public-domain VOGLE graphics package is used as a basis for visualization of the Cowichan Problems. It is available by anonymous FTP from `gondwana.ecr.mu.oz.au::/pub`, and interested readers are referred there for documentation on its internals. Briefly, VOGLE provides routines for creating a window, defining a coordinate system for it, defining a viewing port (clipping window) within it, and for drawing points and simple two-dimensional figures such as lines, boxes, and circles. The details of drawing are all hidden by providing one routine named `gfx_`$x$ for each application $x$. Each such routine takes as its first argument a count of the number of times it has been called; if this count is zero, it initializes the windowing system with appropriate values before invoking lower-level drawing routines. Two separate routines, `gfx_open` and `gfx_close`, are used to open the graphic window initially, and to destroy it when the program terminates. Responsibility is divided in this way so that the same interface routines can be used in both the chained and stand-alone versions of these problems. In all cases, the main driver (either `chain.c` or an application-specific main driver) is responsible only for creating and destroying the window; all other operations are carried out within particular applications. This allows the applications in the chained version to draw inside sub-windows of a single window, while drawing in the whole window when in stand-alone mode.

### 4.2 Input and Output

The specification of the Cowichan Problems allows both formatted and unformatted I/O. Accordingly, the file `io.c` defines two sets of input and output routines, one using formatted read and write operations `fprintf` and `fscanf`, the other using their lower-level (but faster) unformatted equivalents `fwrite` and `fread`. The interfaces to the two sets of routines are isomorphic; a single initialization routine `io_init` is used to set a family of pointers to these functions, so that (for example) all application code can call `io_wrReal2D` to write a real two-dimensional matrix, without having to know whether this refers to `io_wrReal2DFmt` or `io_wrReal2DUnFmt`. For details on the file formats used, please see the accompanying paper.

### 4.3 Scheduling

The file `sch.c` contains support routines to handle work scheduling. At present, the two options are single-axis block distribution, and single-axis cyclic distribution; others may be added in the future. For each scheduling type, a function is defined which takes as input arguments:

- the number of worker processes participating in the operation;
- the ID of the calling worker (in the range 0...NW-1);
- the index of the low end of the loop to be partitioned; and
- the upper bound on the index range of the loop to be partitioned.

The third and fourth parameters were chosen by analogy with the C cliché:

```
for (i=low; i<high; i+=increment){
  loop body
}
```

The fifth, sixth, and seventh parameters of each scheduling routine are pointers to integers, which are filled with:

- the low index of this worker process's portion of the loop;
- the upper limit on this worker process's portion of the loop; and
- the intra-loop stride this worker is to use.

The value returned by the function is a Boolean, which indicates whether or not there is any work for the calling process to do. If *sched* is the name of the scheduling function, its most common use is therefore:

```
if (sched(numThr, ownId, low, high, &start, &end, &stride)){
  for (i=start; i<end; i+=stride){
    loop body
  }
}
```

As with I/O, scheduling functions are usually used indirectly. The initialization functio `sch_init` sets a global pointer `sch_work` to one or the other of the available scheduling functions, so that future calls to `sch_work` invoke the correct behavior. In some contexts, however, the lower-level scheduling functions may be used directly. For example, `randmat` uses the cyclic scheduling function to ensure that the random values it generates are identical to those which are generated in the sequential case.

# 5 Drivers

The main drivers for the stand-alone versions of these applications all follow the same pattern. Each contains definitions of the application-specific variables it requires, followed by argument-handling code. In this, the following flags always have the same meaning:

| | |
|---|---|
| `-g` control | graphics control |
| `-p` scheduling number | parallelism control |
| `-i` name [name] | input file name(s) |
| `-o` name [name] | output file name(s) |
| `-u` | use unformatted files |

The graphics control may be one of `off`, `on`, or `pause`, indicating that graphics is not to be used (the default), that it is to run at full speed, or that the system is to pause after each frame of output. The scheduling control may be one of `block` or `cyclic`; the number given must be positive, and must be no larger than the compile-time parameter `MAXPAR`, which defines the maximum number of concurrent threads. If an input (output) filename is not given where one is needed, the application will read from standard input (output).

The following application-specific flags are also supported:

| | | |
|---|---|---|
| `-E` iters relax | `elastic`: | iterations and relaxation count |
| `-F` fraction | `invperc` and `thresh`: | fraction of matrix to fill |
| `-L` iters | `life`: | iterations to simulate |
| `-M` x0 y0 xe ye | `mandel`: | base point and extent of region |
| `-N` number | `winnow`: | number of points to select |
| `-R` limit seed | `randmat`: | value limit and RNG seed |
| `-S` nrows ncols | `mandel` and `randmat`: | matrix size |
| `-T` tolerance | `sor`: | tolerance on result |

These flags are also used in the chained version of the problem set to control the behavior of individual applications. Two additional flags are also required for this:

| | |
|---|---|
| `-c` xy | path through chained sequence |
| `-d` [stem] | dump intermediate results |

The first flag, `-c`, is used to select a path through the sequence of chained problems. It takes as its argument a two-character code. The right (second) character must be one of `'m'` or `'r'`, to select `mandel` or `randmat`, while the left (first) character must be one of `'i'` or `'t'`, to select invasion percolation or histogram thresholding.

The second flag, `-d`, determines whether or not intermediate results are saved to file or not. If it is not present, intermediate results are not written out; if it is, then files are created using the naming scheme shown in Table 3. The optional argument to this flag specifies a stem, which is used as a prefix when creating file names. Its use is described in Section 9.2.

# 6 Applications

Descriptions of the inputs, outputs, and actions of individual applications can be found in the accompanying paper.

# 7 Multi-Threaded Implementations

Two threaded implementations are provided as a basis for other control-parallel implementations; a C* implementation, which can serve as a basis for other data-parallel implementations, is under construction.

The files `fj/app/app.c` and `bar/app/app.c` contain support routines for creating and synchronizing families of threads, and for marshalling and unmarshalling parameters to these threads. Both sets of routines give each thread in a group of $W$ threads a unique integer identifier in the range $0 \ldots W-1$.

| | |
|---|---|
| `m.i2` | `mandel` |
| `r.i2` | `randmat` |
| `hm.i2` | `half` of `m.i2` |
| `hr.i2` | `half` of `r.i2` |
| `im.b2`, `ir.b2` | `invperc` of `hm.i2`, `hr.i2` |
| `tm.b2`, `tr.b2` | `thresh` of `hm.i2`, `hr.i2` |
| `lim.b2` | `life` of `im.b2` |
| | *path for* `ir.b2`, `tm.b2`, *and* `tr.b2` *omitted* |
| `wim.p1` | `winnow` of `im.b2` and `lim.b2` |
| `nim.p1` | `norm` of `wim.p1` |
| `eim.p1` | `elastic` of `nim.p1` |
| `oim.r1`,`oim.r2` | `outer` of `eim.p1` |
| `gim.r1` | `gauss` of `oim.r2` and `oim.r1` |
| `pgim.r1` | `product` of `oim.r2` and `gim.r1` |
| `sim.r1` | `sor` of `oim.r2` and `oim.r1` |
| `psim.r1` | `product` of `oim.r2` and `sim.r1` |
| `vim.r1` | `vecdiff` of `gim.r1` and `sim.r1` |

Table 3: Standard Output File Names

### 7.1 Repeated Fork-and-Join Implementation

The fork-join implementation uses threads in a fine-grained manner. The "normal" mode of operation is single-threaded and sequential; when a significant piece of work is encountered, the main thread creates a group of threads to do the work co-operatively, and blocks until they have completed. Since the POSIX threads standard requires a function pointer as an entry point for each thread, each chore to be parallelized must be implemented in a separate function. The extra code required to do this accounts for most of the code expansion in this version. To keep parameter marshalling to a minimum, some data structures which are defined within procedures in the serial implementation are re-defined to have file scope in the fork-join version (e.g. the accumulators used in `elastic`).

### 7.2 Barrier-Based Implementation

The barrier implementation uses threads in a coarse-grained manner. The "normal" mode of operation is multi-threaded: a fixed number of threads are created in the main driver, and these repeatedly synchronize using a global barrier as the application executes. The drivers and application implemen-

tations are all modified to pass a unique integer thread ID (called `tid`) to each routine. As in the fork-join version, some data structures which are defined within procedures in the serial implementation are re-defined to have file scope in the fork-join version.

# 8 Support Programs

The directory `aux` contains the source for several support programs, whose executables are placed in the directory `bin`. These programs do the following:

| | |
|---|---|
| `chk` | compare two data files |
| `gen` | generate a data file |
| `sdu` | count disk usage (skipping symbolic links) |
| `slc` | count lines (skipping symbolic links) |
| `view` | display a data file graphically |

# 9 Compilation and Testing

This section describes how to re-build the current implementation, how to generate test output for a single version, and how to compare the output of different versions.

## 9.1 `make` options

The command `make` on its own will re-generate executables for all implementations. It does this by calling itself recursively, specifying the following flags:

| | |
|---|---|
| `root_d` | the root directory for the whole problem set |
| `plat_d` | the directory containing code for a particular platform (version) |
| `plat_f` | flags specific to that platform |
| `plat_l` | auxiliary libraries specific to that platform |
| `gfx_f` | graphics flags |
| `gfx_l` | graphics libraries |
| `app_lf` | the flag specifying the application library for that platform |
| `cmpl` | the compiler to use |
| `obj_f` | the compiler flag to generate an object file |
| `exe_f` | the compiler flag to generate an executable |
| `A` | the suffix used for object archives (usually `.a`) |
| `C` | the suffix used for source code (usually `.c`) |
| `O` | the suffix used for object files (usually `.o`) |

It may seem unnecessarily pedantic to specify such things as file suffices, but some systems require non-standard file identifiers (e.g. `.cs` for C*).

The commands `make tidy` and `make clean` can also be given. Respectively, these delete files that might have been created during work on this code (such as editor backup versions of source files), and delete all files created when building the code (such as libraries and executables).

## 9.2 Testing and Comparing Implementations

The command:

`make tests dir=`*x*

runs all sequences of the Cowichan Problems in both stand-alone and chained form, using a set of small default parameters. The output files are put in the directory *x*/test. Their names following the scheme given in Table 3, with the prefix `A` being used to indicate stand-alone output, and one of `IM`, `IR`, `TM`, or `TR` being used to indicate the output of the chained version along different paths.

The command:

`make test_cmp d_a=`*x* `d_b=`*y*

uses the auxiliary program `chk` to compare the stand-alone files in the directories *x*/test and *y*/test. The output shows the differences between the files, where the "difference" between vectors and matrices of Booleans, integers or reals is defined as the maximum pairwise difference, and the difference between two vectors of points is their greatest pairwise distance. If any of these values is significantly greater than the tolerance value used in the `sor` application, it is very probable that the implementation in question contains a bug.