FORTRAN 90 Modules for Implementation of Parallelised, Model-Independent, Model-Based Processing

John Doherty

Watermark Numerical Computing

March 2008

Table of Contents

1. Introduction

1.1 General

This document describes two modules which can be used to implement application-to-model communication in the same manner as that undertaken by PEST. Thus the programmer of an application which must repeatedly run a model, and process its outputs, is relieved of the task of writing code to implement application-to-model interfacing and the managing of parallel runs across a computer network.

1.2 PEST-Model Interface

As is documented in the PEST manual, PEST must run a model many times in the course of estimating parameters for that model. The "model" can be comprised of a single executable program, or many executable programs encapsulated in a batch or script file. It is required, however, that the model is capable of being run by PEST through the issuing of a single command.

On each occasion that a model is run by PEST, the following steps are taken.

1. First PEST writes one or a number of model input files. Parameters cited within these files are provided with values that PEST wishes the model to employ on that particular run.

2. The model is then run using a system command.

3. Model outputs of interest (which can occur on one or a number of model output files) are read by PEST.

The writing of model input files with updated parameter values is accomplished through the agency of user-supplied templates of these files. In these template files, parameters are named (each name should be 12 characters or less in length), and the spaces to which their current values should be written are identified. There is no limit to the number of locations to which a particular parameter value can be written, nor to the number of parameters that can be employed in this process.

Model output files are read using instructions. One instruction file should be provided for each model output file that must be read. Instructions within each of these files direct the reading of model output files. They also provide names for the model outputs that must be read; these names must be 20 characters or less in length. These outputs are referred to as "observations" in PEST parlance. There is no limit to the number of observations which can be read in this manner. However each observation can only be read once.

For more details of the PEST-to-model interface protocol, see the PEST manual. It should be noted that programs of the USGS JUPITER suite use this same protocol. However the "ptf" and "pif" headers to PEST template and instruction files are replaced by "jtf" and "jif" headers respectively. PEST, as well as the model interface software documented herein, will accept both of these header types. JUPITER programs, however, will only accept the latter

header types.

1.3 Parallelisation of Model Runs

Where the values assigned to parameters for one model run do not depend on the results of the immediately previous model run, great gains in efficiency can be had if packets of model runs are undertaken in parallel. The Parallel PEST version of PEST implements such run parallelisation as it fills the Jacobian matrix of observation sensitivities to parameters. Parameter values pertaining to all model runs which must be undertaken for the purpose of computing finite-difference-based sensitivities are provided to PEST's parallel run manager, which then distributes model runs to computers to which it has access through a network. One or more "slaves" is run on each of those computers; each of these slaves issues the command to run the model when it receives the appropriate signal from PEST. It is assumed that all software and files necessary to run the model are available on each slave machine. Before commanding a slave to undertake a model run, PEST writes input files pertinent to that model run across the network; the writing of these files is governed by one or more template files in the usual manner. After the model run is complete the slave signals PEST of its completed status. PEST then reads pertinent model output files across the network using one or a number of instruction files in the usual manner.

PEST's parallel run manager communicates with slaves through the reading and writing of short message files written to the directory from which each slave is run. It is assumed that these directories can be "seen" from the master computer (on which PEST is run) through the sharing of network drives. The run manager keeps track of which machines can undertake model runs the fastest, preferentially allocating runs to those machines. If at any time a slave drops out of the parallelisation process, this condition is detected, and PEST allocates unfinished or lost model runs to other slaves.

A complete description of PEST's parallelisation capabilities is provided in the PEST manual.

1.4 The Modules Documented Herein

Two FORTRAN 90 modules are documented herein for implementation of PEST-type application-to-model interfacing, and for parallelisation of model runs based on this interfacing. These are designed to be useable by any program which requires that many model runs be undertaken in succession with one or more altered inputs in each case, and that selected outputs of those runs be processed (e.g. for calibration or optimisation purposes). Using these modules, application-to-model communication and parallelisation of model runs can be accomplished using a minimum of subroutine calls, leaving the programmer free to attend to the programming details necessary to implement the algorithm that underpins his/her particular application. It is thereby hoped that availability of the modules documented herein will expedite the writing of useful model-based processing software, and promulgate protocols for implementation of this processing that enhance interchangeability of software.

It is anticipated that the programs documented herein will be improved over time. For example, it is hoped that an option will soon become available to replace the use of message files with MPI-based messaging, this allowing faster and sometimes more stable

communication than can be implemented through the more basic (but more general) message file protocol.

1.5 Source Code

Source code for the model interface and parallelisation modules is found in files *mio.f* and *prm.f* respectively. Two drivers are provided to demonstrate the use of these modules. These are provided in files *driver1.f* and *driver2.f*; *driver1.f* USEs only the first of these modules whereas *driver2.f* USEs both.

Compilation of both the modules and the drivers can be undertaken with the help of the provided *makefile*. As an inspection of the *makefile* reveals, all source code is first processed using the CPPP program which provides very basic emulation of certain compiler preprocessing functionality, normally available through the CPP pre-processor supplied with many C compilers. Source code for CPPP is also provided. CPPP provides the ability to select/deselect fragments of code according to whether symbols supplied on its command line match those supplied with "#ifdef" statements placed within program source code. Selection of appropriate symbols for use in a particular compilation environment takes place on the first line of the *makefile*. This line can be altered according to the platform on which compilation takes place, and according to idiosyncrasies of a user's compiler. Extra symbols can easily be added to the source code (and to the list defined within the *makefile*) to support different compilation options if desired.

As specified through *makefile* directives, pre-processing of *mio.f* and *prm.f* using the CPPP utility produces source code files named *mio.f90* and *prm.f90* These are then ready for compilation and linking to a user's application.

2. MODEL_INPUT_OUTPUT_INTERFACE

2.1 General

Source code for the MODEL_INPUT_OUTPUT_INTERFACE module is contained in file *mio.f*. With the exception of the subroutines documented below, the contents of this module are private. Access to this module is enabled through employing the statement:-

```
USE MODEL_INPUT_OUTPUT_INTERFACE
```

in any subprogram from which any of these subroutines are called.

Public subroutines are now described in the order in which they are likely to be called in a program which USEs this module.

Most of the following subroutines include an *ifail* variable which is returned as positive if an error condition is encountered. The actual error is described in a text string that can be obtained using the MIO_GET_MESSAGE_STRING subroutine. The contents of this string can then be employed by the calling program to report the error.

2.2 Subroutine MIO_INITIALISE

Specifications for subroutine MIO_INITIALISE are as follows.

```
subroutine mio_initialise(ifail,numin,numout,npar,nobs,precision,decpoint)

      implicit none

      integer, intent(out)                     :: ifail
      integer, intent(in)                      :: numin
      integer, intent(in)                      :: numout
      integer, intent(in)                      :: npar
      integer, intent(in)                      :: nobs
      character (len=*), intent(in), optional  :: precision
      character (len=*), intent(in), optional  :: decpoint
```

Subroutine MIO_INITIALISE must be called before any other subroutine of the MODEL_INPUT_OUTPUT_INTERFACE module is called. It provides this module with information that it requires in order to initialise arrays for storage of its variables.

Arguments are as follows.

Argument	Description
IFAIL	This is returned as zero unless an error condition is encountered. In this event the pertinent error message string can be retrieved using the MIO_GET_MESSAGE_STRING subroutine.
NUMIN	NUMIN is the number of template files required for the application-to-model interface. Each template file is employed to write a different model input file prior to running the model. NUMIN must be 1 or greater.

NUMOUT	NUMOUT is the number of instruction files employed in the application-to-model interface. Each instruction file is used to read a different model output file after the model has run. NUMOUT must be 1 or greater.
NPAR	NPAR is the number of different parameters cited in all model template files. Note that the same parameter can be cited many different times on one or more template files. However that parameter is provided with only one value for all of its occurrences.
NOBS	NOBS is the number of different observations read from model output files using instruction files. These are provided with names through the instructions which read them.
PRECISION	This must be supplied as “single” or “double”. In the former case single precision protocol is observed in writing numbers to model input files; in the latter case double precision protocol is observed. In the former case a maximum of 13 spaces is used to represent a number; in the latter case a maximum of 23 spaces is used to represent a number. See the PEST manual for more details. Default is “single”.
DECPOINT	This must be supplied as “point” or “nopoint”. If it is supplied as “nopoint” parameter values will be written in a format whereby the decimal point is omitted if possible. This is useful where parameter spaces in template files are small. See the PEST manual for more details. Default is “point”.

2.3 Subroutine MIO_PUT_FILE

This subroutine must be called after MIO_INITIALISE and before any other MODEL_INPUT_OUTPUT_INTERFACE subroutines are called. It is used to inform the MODEL_INPUT_OUTPUT_INTERFACE module of the names of all template files and corresponding model input files, and of all instruction files and corresponding model output files. It must be called 2×NUMIN + 2×NUMOUT times.

Specifications for subroutine MIO_PUT_FILE are as follows.

```
subroutine mio_put_file(ifail,itype,inum,filename)

       integer, intent(out)            :: ifail
       integer, intent(in)             :: itype
       integer, intent(in)             :: inum
       character (len=*), intent(in)   :: filename
```

Argument descriptions are provided in the table below.

Argument	**Description**
IFAIL	This is returned as zero unless an error condition is encountered. In this event the pertinent error message string can be retrieved using the MIO_GET_MESSAGE_STRING subroutine.

ITYPE	The type of file whose name is being supplied:- 1 – template file; 2 – model input file; 3 – instruction file; 4 – model output file.
INUM	Each template file must be matched to a model input file; each instruction file must be matched to a model output file. There are NUMIN pairs of the former and NUMOUT pairs of the latter. INUM indicates to MIO_PUT_FILE the pair number to which the supplied filename belongs. INUM must be greater than zero. It can be no greater than NUMIN if a template or model input filename is supplied, and no greater than NUMOUT if an instruction or model output filename is supplied.
FILENAME	The name of a template file, model input file, instruction file or model output file. The name of this file must be no greater than 200 characters in length.

2.4 Subroutine MIO_GET_FILE

It is possible that this subroutine will not be called in an application that USEs the MODEL_INPUT_OUTPUT_INTERFACE module. It is provided for convenience. Its specifications are as follows.

```
subroutine mio_get_file(ifail,itype,inum,filename)
       integer, intent(out)             :: ifail
       integer, intent(in)              :: itype
       integer, intent(in)              :: inum
       character (len=*), intent(out)   :: filename
```

The arguments employed by MIO_GET_FILE are the same as those employed by MIO_PUT_FILE. See the description of that subroutine for details.

2.5 Subroutine MIO_PROCESS_TEMPLATE_FILES

Subroutine MIO_PROCESS_TEMPLATE_FILES must not be called before the names of all template and model input files are provided to the MODEL_INPUT_OUTPUT_INTERFACE module through calls to subroutine MIO_PUT_FILE. It must be called before the first model run is undertaken. It checks all template files for correctness and consistency. If an error is encountered this is recorded internally and IFAIL is returned as 1. The error message can be subsequently retrieved using the MIO_GET _MESSAGE_STRING subroutine.

Subroutine MIO_PROCESS_TEMPLATE_FILES should be called only once by an application which USEs the MODEL_INPUT_OUTPUT_INTERFACE module.

Specifications are as follows.

```
subroutine mio_process_template_files(ifail,npar,apar)
        integer, intent(out)                 :: ifail
        integer, intent(in)                  :: npar
        character (len=*), dimension(npar)   :: apar
```

Argument descriptions are set out in the table below.

Argument	Description
IFAIL	This is returned as zero unless an error condition is encountered. In this event, the error message string can be retrieved using the MIO_GET_MESSAGE_STRING subroutine.
NPAR	The number of parameter names supplied through the APAR array.
APAR	A character array of NPAR elements containing parameter names. Each name cited in a template file must be cited in this array. However if this array contains parameter names that are not cited in any template files, this will not be construed as an error condition. Parameter names must be 12 characters or less in length. **Note that parameter names must be supplied in lower case.**

2.6 Subroutine MIO_STORE_INSTRUCTION_SET

Subroutine MIO_STORE_INSTRUCTION_SET must not be called before the names of all instruction and corresponding model output files have been provided to the MODEL_INPUT_OUTPUT_INTERFACE module through calls to subroutine MIO_PUT_FILE. It must be called before the first model run is undertaken. It checks all instruction files for correctness and consistency, and stores instructions contained therein in compressed form for later fast retrieval during the actual reading of model output files. If an error is encountered this is recorded internally and IFAIL is returned as 1. The error message can subsequently be retrieved using the GET_MIO_MESSAGE_STRING subroutine.

Subroutine MIO_STORE_INSTRUCTION_SET should be called only once by a program which USEs the MODEL_INPUT_OUTPUT_INTERFACE module.

Specifications are as follows.

```
subroutine mio_store_instruction_set(ifail)

        integer, intent(out)  :: ifail
```

The description of its single argument follows.

Argument	Description
IFAIL	This is returned as zero unless an error condition is encountered. In this event the pertinent error message string can be retrieved using the MIO_GET_MESSAGE_STRING subroutine.

2.7 Subroutine MIO_DELETE_OUTPUT_FILES

As is described in the PEST manual, all model output files should be deleted before the model is run. Thus if the model fails to run, an old model output file will not be mistaken for the latest one.

Its specifications are as follows.

```
subroutine mio_delete_output_files(ifail,asldir)

        integer, intent(out)                :: ifail
        character(*), intent(in), optional  :: asldir
```

Argument descriptions are provided in the table below.

Argument	Description
IFAIL	This is returned as zero unless an error condition is encountered. In this event the pertinent error message string can be retrieved using the MIO_GET_MESSAGE_STRING subroutine.
ASLDIR	The name of the subdirectory in which model input files are found. This name is appended to the front of each model input filename. On a PC it must conclude with a "\" character and on a UNIX system it must conclude with a "/" character. The (optional) ASLDIR argument is not normally used. It is employed by the PARALLEL_RUN_MANAGER module however, in a way that is invisible to the user.

2.8 Subroutine MIO_WRITE_MODEL_INPUT_FILES

This subroutine should be called prior to every model run. It instructs the MODEL_INPUT_OUTPUT_INTERFACE module to write a set of model input files in which parameters are provided with values supplied through one of its arguments. It must **not** be called prior to subroutine MIO_PROCESS_TEMPLATE_FILES. However once the latter is called, MIO_WRITE_MODEL_INPUT_FILES can be called as many times as desired. Presumably parameter values will be different on each such call.

Specifications are as follows:-

```
subroutine mio_write_model_input_files(ifail,npar,apar,pval,asldir)

        integer, intent(out)                                :: ifail
        integer, intent(in)                                 :: npar
        character (len=*), intent(in), dimension(npar)      :: apar
        double precision, intent(inout), dimension(npar)    :: pval
        character (len=*), intent(in), optional             :: asldir
```

Argument	Description
IFAIL	This is returned as zero unless an error condition is encountered. In this event the pertinent error message string can be retrieved using the MIO_GET_MESSAGE_STRING subroutine.
NPAR	The number of parameter names supplied through the APAR array.

APAR	A character array of NPAR elements containing parameter names. Each name cited in a template file must be cited in this array. However if this array contains parameter names that are not cited in any template files, this will not be construed as an error condition. Parameter names must be 12 characters or less in length. **Parameter names must be supplied in lower case.** They need not be supplied in the same order as in the previous call to subroutine MIO_PROCESS_TEMPLATE_FILES or in other calls to MIO_WRITE_MODEL_INPUT FILES. Also, parameter names provided in the current call to subroutine MIO_WRITE_MODEL_INPUT_FILES can be different from those employed on other calls to this subroutine, provided no names that occur on template files are omitted in either case.
PVAL	This is an array of parameter values. It is presumed that elements are provided in the same order as in the APAR array, thus allowing parameter values to be linked to parameter names. Like PEST, the MODEL_INPUT_OUTPUT_INTERFACE module may make slight adjustments to parameter values if they cannot be written to model input files with the same precision as that with which they are recorded internally by the calling program. Thus the internal and external representations of these numbers become identical. (The main program should thus read these values back from the PVAL array.) This inhibits the occurrence of roundoff errors in finite-difference derivatives calculation.
ASLDIR	The name of the subdirectory in which model input files are found. This name is appended to the front of each model input filename. On a PC it must conclude with a "\" character and on a UNIX system it must conclude with a "/" character. The (optional) ASLDIR argument is not normally used. It is employed by the PARALLEL_RUN_MANAGER module however, in a way that is invisible to the user.

2.9 Subroutine MIO_READ_MODEL_OUTPUT_FILES

Subroutine MIO_READ_MODEL_OUTPUT_FILES should be called after every model run. It reads model output files using the instructions provided in the previously-stored instruction set. Thus it must only be called **after** subroutine MIO_STORE_INSTRUCTION_SET has been called.

Specifications are as follows:-

```
subroutine
mio_read_model_output_files(ifail,nobs,aobs,obs,instruction,asldir)

        integer, intent(out)                              :: ifail
        integer, intent(in)                               :: nobs
        character (len=*), intent(in), dimension(nobs)    :: aobs
        double precision, intent(out), dimension(nobs)    :: obs
        character (len=*), intent(out)                    :: instruction
        character (len=*), optional                       :: asldir
```

Argument descriptions are presented in the table below.

Argument	**Description**

IFAIL	This is returned as zero unless an error condition is encountered. In this event the pertinent error message string can be retrieved using the MIO_GET_MESSAGE_STRING subroutine. If appropriate, the offending instruction will be provided in the INSTRUCTION string.
NOBS	The number of observation names supplied through the AOBS array.
AOBS	A character array of NOBS elements containing observation names. Each name cited in an instruction file must be cited in this array. Conversely, any name cited in the AOBS array must be cited in an instruction file. Observation names must be 20 characters or less in length. **Observation names must be supplied in lower case.** They need not be supplied in the same order as in other calls to MIO_READ_MODEL_OUTPUT_FILES.
OBS	This is an array of observation values read from model output files. It is presumed that elements are referenced in the same order as for the AOBS array; it is through this mechanism that observation values are linked to observation names.
INSTRUCTION	If an error condition occurs as a result of an incorrect instruction, or because a certain instruction is not able to read the expected number from a model output file, IFAIL is returned as 1. As usual, an error message is recorded, which can then be retrieved through the MIO_GET_MESSAGE_STRING subroutine. The offending instruction is recorded in the INSTRUCTION variable. Thus a program which calls MIO_READ_MODEL_OUTPUT_FILES should check whether the INSTRUCTION variable is not empty after an error condition is encountered (and IFAIL is therefore returned as positive). The offending instruction can then be displayed together with the reason for the error as provided by the MIO_GET_MESSAGE_STRING subroutine.
ASLDIR	The name of the subdirectory in which model output files are found. This name is appended to the front of each model output filename. On a PC it must conclude with a "\" character while on a UNIX system it must conclude with a "/" character. The (optional) ASLDIR argument is not normally used. It is employed by the PARALLEL_RUN_MANAGER module however, in a way that is invisible to the user.

2.10 Subroutine MIO_GET_MESSAGE_STRING

If an error condition occurs, as indicated by a positive IFAIL value being returned from a particular subroutine, the reason for the error will be recorded in a character string which can be retrieved using the MIO_GET_MESSAGE_STRING subroutine.

Specifications for MIO_GET_MESSAGE_STRING are as follows:-

```
subroutine mio_get_message_string(ifail,amessage_out)

      integer, intent(out)       :: ifail
      character*(*), intent(out) :: amessage_out
```

Arguments are described in the following table.

Argument	**Description**
IFAIL	This is returned as zero unless an error condition is encountered.
AMESSAGE_OUT	A character variable containing the error message. This message will never be more than 500 characters in length.

2.11 Subroutine MIO_FINALISE

Subroutine MIO_FINALISE should be called prior to termination of the program which USEs the MODEL_INPUT_OUTPUT_INTERFACE module. It de-allocates all memory allocated by this module. Its specifications are as follows.

```
subroutine mio_finalise(ifail)

      integer, intent(out) :: ifail
```

The description of its single argument follows.

Argument	**Description**
IFAIL	This is returned as zero unless an error condition is encountered. In this event the pertinent error message string can be retrieved using the MIO_GET_MESSAGE_STRING subroutine.

2.12 Subroutine MIO_GET_STATUS

Subroutine MIO_GET_STATUS is rarely needed. It is provided as a convenience. Its specifications are as follows.

```
subroutine mio_get_status(template_status,instruction_status)

      integer, intent(out)  :: template_status
      integer, intent(out)  :: instruction_status
```

MIO_GET_STATUS arguments are described in the following table.

Argument	**Description**
TEMPLATE_STATUS	This is returned as 1 if subroutine MIO_PROCESS_TEMPLATE_FILES has been previously called. Otherwise it is returned as 0.
INSTRUCTION_STATUS	This is returned as 1 if subroutine MIO_STORE_INSTRUCTION_SET has been previously called. Otherwise it is returned as 0.

2.13 Subroutine MIO_GET_DIMENSIONS

Subroutine MIO_GET_DIMENSIONS is rarely needed. It is provided as a convenience. Its specifications are as follows.

```
subroutine mio_get_dimensions(numinfile,numoutfile)

      integer, intent(out)  :: numinfile
      integer, intent(out)  :: numoutfile
```

MIO_GET_DIMENSIONS arguments are described in the following table.

Argument	**Description**
NUMINFILE	The number of model input files. This is equivalent to NUMIN as previously supplied through subroutine MIO_INITIALISE.
NUMOUTFILE	The number of model output files. This is equivalent to NUMOUT as previously supplied through subroutine MIO_INITIALISE.

3. PARALLEL_RUN_MANAGER

3.1 General

Source code for the PARALLEL_RUN_MANAGER module is found in file *prm.f*. With the exception of the subroutines documented below, the contents of this module are private. Access to this module is enabled through employing the statement:-

```
USE PARALLEL_RUN_MANAGER
```

in any subprogram from which any of these subroutines are called.

Public subroutines are described below in the order in which they are likely to be called in a program which USEs this module. Most of them include an *ifail* argument; this is returned as positive if an error condition is encountered. The error is described in one or more text strings that can be obtained using the PRM_GET_MESSAGE_STRINGS subroutine. If desired, these strings can be used directly for error reporting by the calling program.

3.2 Interaction with MODEL_INPUT_OUTPUT_INTERFACE Module

The PARALLEL_RUN_MANAGER module USEs the MODEL_INPUT_OUTPUT_INTERFACE module for all communications with model input and output files residing in different slave directories. The MODEL_INPUT_OUTPUT_INTERFACE module should thus be initialised, and certain of its subroutines called, before the PARALLEL_RUN_MANAGER module is initialized. In particular, it is the responsibility of the calling program to run the following MODEL_INPUT_OUTPUT_INTERFACE subroutines before initialising the PARALLEL_RUN_MANAGER module.

MIO_INITIALISE

MIO_PUT_FILE

MIO_PROCESS_TEMPLATE_FILES

MIO_STORE_INSTRUCTION_SET

It is also the user's responsibility to run the MIO_FINALISE subroutine when all parallel runs have been completed.

Note that MIO_PUT_FILE must be run for all template, model input, instruction and model output files. When calling this subroutine, there is no need to consider the parallelisation of model runs. Hence the names of all model input and output files are provided to this subroutine as if they reside on the master machine and no parallelisation is to take place at all. The PARALLEL_RUN_MANAGER module will construct the names of the model input files that it will actually write and the model output files that it will actually read when model runs are undertaken on slave machines by affixing the name of a subdirectory to the front of the nominated model input and output files. This is assumed to be the same subdirectory as

that to which it writes message files for pertinent slaves (i.e. the working directory of these slaves). Note that this convention does not preclude the ability of a model to read/write its input/output files from/to different subdirectories on each slave machine on which it resides. Relativity of filenames as seen from each slave machine can still be preserved by proper definition of model input/output filenames, these names including a path relative to the slave working directory. The subdirectory name through which a particular set of these different model input and output files can be distinguished from another set when seen from the master directory can then be prefixed to these names by the PARALLEL_RUN_MANAGER module.

3.3 Parallelisation Principles

Parallelisation of model runs is undertaken in the same manner as is undertaken by Parallel PEST. Thus the program which USEs the PARALLEL_RUN_MANAGER module must reside on a "master machine" while model runs are conducted on "slave machines". The same slaves are employed on these machines as are employed by Parallel PEST. These are named PSLAVE. They are each run by typing "pslave" at the command-line prompt while situated within the respective slave working directory. (This will mostly be the same as the working directory of the model that is run by the respective slave.) Upon commencement of execution each slave prompts for the command that it must issue in order to run the model. It then issues this command only when instructed to do so by the PARALLEL_RUN_MANAGER module residing on the master machine. See the PEST manual for more details.

When the PARALLEL_RUN_MANAGER module is initialised, it first attempts to communicate with all nominated slaves. If it cannot find all of them, it commences the parallelised model run process anyhow, anticipating that missing slaves will appear later. As they are detected, they are put to work in the carrying out of model runs.

3.4 Subroutine PRM_INITIALISE

As the name suggests, subroutine PRM_INITIALISE initiates execution of the PARALLEL_RUN_MANAGER module. It should be called only once. Its specifications are as follows.

```
subroutine
prm_initialise(ifail,prm_mr,prm_mf,prm_wk,prm_nr,nslave,maxrun,iwait,repeat
run)

      integer, intent(out)      :: ifail
      integer, intent(in)       :: prm_mr
      integer, intent(in)       :: prm_mf
      integer, intent(in)       :: prm_wk
      integer, intent(in)       :: prm_nr
      integer, intent(in)       :: nslave
      integer, intent(in)       :: maxrun
      integer, intent(in)       :: iwait
      integer, intent(in)       :: repeatrun
```

The role of each of the arguments employed by PRM_INITIALISE is provided in the following table.

Argument	Description

IFAIL	This is returned as zero unless an error condition is encountered. In this event the pertinent error message string can be retrieved using the PRM_GET_MESSAGE_STRINGS subroutine.
PRM_MR	The unit number of the parallel run management record file. The PARALLEL_RUN_MANAGER module writes a record to this file of all communications between the manager and its slaves. **This file should be open prior to calling subroutine PRM_INITIALISE.**
PRM_MF	A unit number that module PARALLEL_RUN_MANAGER can use for work files. **This unit number should not be employed for any purpose in the calling program.**
PRM_WK	A unit number that module PARALLEL_RUN_MANAGER can use for work files. **This unit number should not be employed for any purpose in the calling program.**
PRM_NR	If supplied with a positive value, the PARALLEL_RUN_MANAGER module records the completion of each requested run to a file with this unit number. If required (i.e. if PRM_NR is positive), **the file should be opened by the calling program before subroutine PRM_INITIALISE is called.**
NSLAVE	The number of slaves to which the PARALLEL_RUN_MANAGER module has access.
MAXRUN	The maximum number of runs that will ever be required in any parallelised run package.
IWAIT	The PARALLEL_RUN_MANAGER module undertakes periodic strategic pauses in communications with slave machines in order to avoid the crossing of operating system messages regarding the status of files which are handled by the module (including message files and model input/output files). IWAIT is the length of each such strategic stoppage in one hundredths of a second. A value of 20 is suggested; however this should be increased if “file access” errors are encountered.
REPEATRUN	If a model output file cannot be read, the PARALLEL_RUN_MANAGER module will immediately cease execution with an appropriate error message if REPEATRUN is set to 0. If it is set to 1 however, three attempts will be made to repeat the model run which caused the problem – possibly on different slave machines - before an error condition is reported.

3.5 Subroutine PRM_SLAVEDAT

A program which USEs the PARALLEL_RUN_MANAGER module supplies this module with information about the slaves to which it has access through the PRM_SLAVEDAT subroutine. This must be called after the PRM_INITIALISE subroutine and before any other subroutines of the PARALLEL_RUN_MANAGER module are called. It must be called once for each slave.

PRM_SLAVEDAT will report an error condition if calls have not been made in the main

program to subroutines MIO_PROCESS_TEMPLATE_FILES and MIO_STORE_INSTRUCTION_SET. Thus PRM_SLAVEDAT expects to find that all data input and checking required by the MODEL_INPUT_OUTPUT_INTERFACE module has been performed.

Specifications for subroutine PRM_SLAVEDAT are as follows.

```
subroutine prm_slavedat(ifail,islave,iruntme,aslave,asldir)

      integer, intent(out) :: ifail
      integer, intent(in)  :: islave
      integer, intent(in)  :: iruntme
      character (len=*), intent(in)  :: aslave
      character (len=*), intent(in)  :: asldir
```

The role of each of its arguments is set out in the following table.

Argument	**Description**
IFAIL	This is returned as zero unless an error condition is encountered. In this event, the pertinent error message string(s) can be retrieved using the PRM_GET_MESSAGE_STRINGS subroutine.
ISLAVE	The slave number for which information is supplied through the current PRM_SLAVEDAT call.
IRUNTME	The expected model run time on the nominated slave machine in seconds. It is best to err on the side of long run times when supplying this variable; see below.
ASLAVE	The name of a slave. This can be any string of up to 35 characters in length.
ASLDIR	The working directory of the slave machine as seen from the master machine. If working on a PC this must end in "\"; if working on a UNIX machine it must end in "/". This name is prefixed to the names of all model input files and all model output files as supplied through calls to the MIO_PUT_FILE subroutine when writing/reading model data for this particular slave. It is also prefixed to the names of all message files used for communication with the nominated slave.

3.6 Subroutine PRM_SLAVETEST

Subroutine PRM_SLAVETEST must not be called until information is provided for all slaves through a series of PRM_SLAVEDAT calls. However it must be called before any calls are made to the PRM_DORUNS subroutine. PRM_SLAVETEST ensures that the PARALLEL_RUN_MANAGER module can write to all slave working directories and read data from those same directories. It then tests for the presence of all slaves. If no slaves are found it reports an error condition to the calling program.

Specifications for subroutine PRM_SLAVETEST are as follows.

```
subroutine prm_slavetest(ifail)

      integer, intent(out) :: ifail
```

A description of its single argument is provided in the following table.

Argument	Description
IFAIL	This is returned as a positive number if an error condition is encountered. The error message string(s) can then be retrieved using the PRM_GET_MESSAGE_STRINGS subroutine. IFAIL is returned as a negative number if program stoppage has been activated through the presence of a *pest.stp* file. In this case it returns -1 if the integer present in file *pest.stp* is 1, or -2 if the integer recorded in file *pest.stp* is 2. See below for more information on stopping and starting. In all other cases a value of zero is returned for IFAIL.

3.7 Subroutine PRM_DORUNS

This is the work-horse of the PARALLEL_RUN_MANAGER module. It dispatches a set of model runs to its slaves, and monitors the completion of those runs. In the meantime if any absent slaves appear, these are immediately put to work. If excessive model run times indicate that one or more slaves have disappeared, it re-assigns pertinent model runs to different slaves. When all allocated model runs are complete, it returns control to the calling program.

Specifications for subroutine PRM_DORUNS are as follows.

```
subroutine prm_doruns(ifail,itn,npar,nobs,nrun,pregdim,pobsdim,parreg,
                      obsreg,apar,aobs,irestart,restartfile)

       integer, intent(out)    :: ifail
       integer, intent(in)     :: itn
       integer, intent(in)     :: npar
       integer, intent(in)     :: nobs
       integer, intent(in)     :: nrun
       integer, intent(in)     :: pregdim
       integer, intent(in)     :: pobsdim

       double precision, intent(inout) :: parreg(pregdim,nrun)
       double precision, intent(inout) :: obsreg(pobsdim,nrun)

       character (len=*), intent(in)   :: apar(npar)
       character (len=*), intent(in)   :: aobs(nobs)

       integer, intent(inout), optional        :: irestart
       character (len=*), intent(in), optional :: restartfile
```

Argument details are provided in the following table.

Argument	Description

IFAIL	This is returned as a positive number if an error condition is encountered. The error message string(s) can then be retrieved using the PRM_GET_MESSAGE_STRINGS subroutine. It is returned as a negative number if program stoppage has been activated through the presence of a *pest.stp* file. In this case it returns -1 if the integer present in file *pest.stp* is 1, or -2 if the integer recorded in file *pest.stp* is 2. See below for more information on stopping and starting. In all other cases it returns a value of zero.
ITN	The parallel run package number. This number is recorded in the run record file written by the PARALLEL_RUN_MANAGER package.
NPAR	The number of entries in the APAR array.
NOBS	The number of entries in the AOBS array.
NRUNS	The number of model runs comprising the current parallel run package.
PREGDIM	Leading dimension of the PARREG array.
POBSDIM	Leading dimension of the OBSREG array.
PARREG	Each column of this matrix contains parameter values to be used on a particular parallel run. These must be supplied in the same order as parameter names in the APAR array. If more parameters are named than are featured in template files for the current problem, their values are simply ignored. Like PEST, the PARALLEL_RUN_MANAGER module may make slight adjustments to parameter values if they cannot be written to model input files with the same precision as that with which they are stored internally by the calling program in order that the internal and external representations of these numbers thereby become identical. The main program should thus read these values back from the PARREG array. This suppresses roundoff errors in finite-difference derivatives calculation.
OBSREG	Each column of this matrix is filled with observation values read from model output files written on a particular parallel run. The ordering of observations is the same as that in which observation names are supplied in the AOBS array. The ordering of columns in this array (each pertaining to a separate model run) is the same as the ordering of model runs supplied in the PARREG array. Thus parameter values in the latter array and observation values in the former array are linked by column number. If an observation name is supplied in the AOBS array and an instruction has not been provided to read that observation, an error condition is decreed to have occurred.
APAR	Parameter names. Each parameter name must be 12 characters or less in length. Any parameter that is cited in any template file whose name was previously provided to the MODEL_INPUT_OUTPUT_INTERFACE module by the main program must be named in this array. **All parameter names must be provided in lower case.**

AOBS	Observation names. Each observation name provided in this array must be cited in an instruction file whose name was provided to the MODEL_INPUT_OUTPUT_INTERFACE module by the main program. **All observation names must be provided in lower case.**
IRESTART	This should be supplied as 1 if a restart file is to be written based on the current parallel run package. It should be supplied as 2 if the current parallel run package should retrieve whatever results are available from an existing restart file, and then undertake the remainder of the model runs requested by the package. (In this case the PARALLEL_RUN_MANAGER module will continue to store run results in a restart file as continuation of the parallel run process takes place). In either of these cases the name of the restart file should be provided through the RESTARTFILE variable. Alternatively, if no restart data storage or retrieval is wanted, IRESTART should be supplied as zero, or simply omitted. If IRESTART is provided as 2, it is returned as 1 to prevent restart data from being read when the next parallel run package is requested. Immediately after subroutine DO_RUNS reads restart data, it appends information to the restart file pertaining to new model runs so that the file is available for subsequent restarts, in accordance with the revised IRESTART setting of 1.
RESTARTFILE	The name of the binary restart file. This is only required if IRESTART is supplied as 1 or 2.

3.8 Subroutine PRM_GET_MESSAGE_STRINGS

If an error condition is encountered on a call to any PARALLEL_RUN_MANAGER subroutine, a text string (or multiple strings) describing the error can be retrieved through the PRM_GET_MESSAGE_STRINGS subroutine. These can then be used in formulation of an error message by the calling program. Note that if more than one message string must be retrieved for description of an error message, each of these strings should commence on a new line when that error message is displayed by the calling program.

Specifications for subroutine PRM_GET_MESSAGE_STRINGS are as follows:-

```
subroutine prm_get_message_strings(ifail,numused,amessage,suppl_amessage)

     integer, intent(out)                          :: ifail
     integer, intent(out)                          :: numused
     character (len=*), intent(out)                :: amessage
     character (len=*), dimension(:),intent(out)   :: suppl_amessage
```

Subroutine argument details are provided in the following table.

Argument	**Description**
IFAIL	This is returned as 1 if an error condition is encountered.

NUMUSED	The number of error message strings required for the current message. If NUMUSED is returned as 1 (as is normally the case) then the entire error message is provided in the AMESSAGE string. If it is returned as 2, then the error message is supplied in two parts, the first being in the AMESSAGE string and the second being in the first element of the SUPPL_AMESSSAGE string array. If it is returned as 3, then AMESSAGE and both strings comprising the SUPPL_AMESSAGE array contain error message strings.
AMESSAGE	Error message string. This will never need to be more than 500 characters in length.
SUPPL_AMESSAGE	Two supplementary error message strings. They will never need to be more than 500 characters in length.

3.9 Subroutine PRM_SLAVESTOP

This subroutine writes a message file to all slave working directories, instructing each slave to cease execution. Thus the user does not need to shut down these slaves him/herself.

Specifications are as follows:-

```
subroutine prm_slavestop(ifail)
```

Details of the single PRM_SLAVESTOP argument are provided in the following table.

Argument	**Description**
IFAIL	IFAIL is returned as non-zero only if an error condition is encountered. A description of the error can then be obtained through a call to subroutine PRM_GET_MESSAGE_STRINGS.

3.10 Subroutine PRM_FINALISE

This subroutine should be the final call made to any subroutine of the PARALLEL_RUN_MANAGER module by the calling program. All arrays employed by the PARALLEL_RUN_MANAGER module are deallocated in this subroutine.

Specifications are as follows.

```
subroutine prm_finalise(ifail)
```

Details of the single PRM_FINALISE argument are provided in the following table.

Argument	**Description**

IFAIL	IFAIL is returned as non-zero only if an error condition is encountered. A description of the error can then be ascertained through a call to subroutine PRM_GET_MESSAGE_STRINGS.

3.11 The Parallel Run Management Record File

The PARALLEL_RUN_MANAGER module writes a record of all communications between it and its slaves to a file whose unit number is provided by the user through subroutine PRM_INITIALISE. This file must have been opened by the main program prior to the calling of subroutine PRM_INITIALISE. It is also the responsibility of the main program to close this file after it has ceased using the PARALLEL_RUN_MANAGER module.

3.12 Model Run Times

The PARALLEL_RUN_MANAGER module keeps track of the time that is required for the model to run on different slave machines. Runs are allocated preferentially to the fastest available slave. Furthermore, if the module finds that it is waiting for a model run to finish for a significantly longer time that that which it is expected to take, the PARALLEL_RUN_MANAGER module will presume that the slave has been lost, and will re-allocate the lost run to a different slave.

Before any model runs have actually been undertaken the PARALLEL_RUN_MANAGER module must rely on user estimates of model run times supplied through the PRM_SLAVEDAT subroutine in order to make run allocation (and possibly run re-allocation) decisions. An undesirable consequence of supplying model run times which are too low, is that the run manager may decide that a run is overdue when it isn't actually overdue at all. The run may then be re-commenced on a spare machine. That machine will not be available to undertake any further model runs (even if separate parallel run packages are initiated on later PRM_DORUNS calls) until that run is finished. This may result in unnecessary delays in overall parallel run processing. Hence it is best to over-estimate rather than under-estimate model run times when supplying them to the PRM_SLAVEDAT subroutine.

(Note that as soon as the first model run has been completed, the PARALLEL_RUN_MANAGER module updates estimates of **all** model run times, using a correction factor which relates estimated time to actual time required to complete this first model run.)

3.13 Stopping and Pausing

Like PEST, the PARALLEL_RUN_MANAGER module periodically checks for the presence of a file named *pest.stp*. As recorded in the PEST manual, this file should contain only a single entry. An entry of 1 or 2 constitutes a message to cease execution immediately (respectively with or without a statistical printout in the case of PEST). An entry of 3 is an instruction to pause execution, while an entry of 4 constitutes an instruction to re-commence paused execution. This file can be written using a text editor, or by another program. It can

also be written using the PSTOP, PSTOPST, PPAUSE and PUNPAUSE utilities provided with PEST.

Both of subroutines PRM_SLAVETEST and PRM_DORUNS check continuously for the presence of file *pest.stp*. If this file is found, the response of the PARALLEL_RUN_MANAGER is as follows.

If the integer contained in *pest.stp* is 3, then the module will simply pause execution, taking no further action until *pest.stp* is re-written with a 4 replacing the 3. Then it will resume execution as if nothing had happened. (Model runs are unaffected however.)

If the integer contained in *pest.stp* is 1 or 2, then execution of subroutine PRM_SLAVETEST or PRM_DORUNS is immediately halted and control is returned to the main program. (Models will continue to run on slave machines until they reach their natural completion; however the results will not be read by the PARALLEL_RUN_MANAGER module.) The IFAIL value returned by the pertinent PARALLEL_RUN_MANAGER subroutine will be -1 or -2, for *pest.stp* entries of 1 or 2 respectively. The calling program can undertake whatever action it wishes on the basis of these returned values. Note that no message string is written when subroutine execution is terminated in this manner.

3.14 Restarting

If directed to do so through the IRESTART variable, subroutine PRM_DORUNS will store elements of the OBSREG array to a binary file as they are filled on completion of successive model runs. If execution of the PARALLEL_RUN_MANAGER module is halted prematurely (for example using the PSTOP or PSTOPST commands, or by simply pressing Ctl-C), retrieval of the contents of this file will obviate the need for already-completed model runs to be undertaken again. Instead, on re-commencement of execution, the outcomes of these runs can simply be read from the restart file, and yet-to-be-completed runs commenced as if nothing had happened; this functionality is activated by setting IRESTART to 2 on the first call to PRM_DORUNS made by the application that USEs the PARALLEL_RUN_MANAGER module.

Caution should be exercised in restarting parallel run execution in this fashion. PRM_DORUNS will not restart a parallel run package unless the parallel run package number provided through its ITN argument agrees with that recorded in the restart file. If this condition is not met, it returns control to the main program with IFAIL set to 1, and with a description of this condition recorded in its error message string.

It follows that if a program that calls the PARALLEL_RUN_MANAGER module wishes to take advantage of restart functionality provided by this module, it must keep track of where in its own operations a previous run interruption occurred. This means that it must store in its own restart file the “state-of-play” of its own algorithm up to that point of processing at which the previous package of parallel model runs had been completed. Upon restart, it should be capable of returning to exactly that point of its own processing sequence before issuing the same call to PRM_DORUNS, with the same ITN argument, as that which it employed when parallel run management was prematurely halted on its previous run.

4. Two Drivers

4.1 General

Two drivers have been provided which illustrate the use of the MODEL_INPUT_OUTPUT_INTERFACE and PARALLEL_RUN_MANAGER modules. Source code for these drivers is in files *driver1.f* and *driver2.f* respectively. See the *makefile* for compilation details. As for the above two modules, preprocessing of the *driver1.f* and *driver2.f* files by the CPPP utility is first required in order to produce files *driver1.f90* and *driver2.f90* which are ready for compilation.

Each of these programs is now described in detail. Note that these programs are provided for demonstration purposes only. Error checking and other functionality provided in each of them is limited.

4.2 The DRIVER1 Utility

DRIVER1 undertakes a series of model runs using parameter values supplied by the user. It USEs only the MODEL_INPUT_OUTPUT_INTERFACE module, and therefore does not undertake parallelised model runs. Its execution is initiated by typing the command “driver1” at the screen prompt. It then prompts for the name of an input file and for the name of an output file.

A typical DRIVER1 input file is illustrated below.

```
* control data
5 19
2 3
* parameter data
ro1  4.0
ro2  10.0
ro3  1.0
h1   1.5
h2   10.0
* observation data
ar1
ar2
ar3
ar4
ar5
ar6
ar7
ar8
ar9
ar10
ar11
ar12
ar13
ar14
ar15
ar16
ar17
ar18
ar19
* model command line
ves
* model input/output
ves1.tpl a_model.in1
ves2.tpl a_model.in2
ves1.ins a_model.ot1
ves2.ins a_model.ot2
ves3.ins a_model.ot3
```

A DRIVER1 input file.

Like the PEST control file, the DRIVER1 input file is divided into sections. Each section is identified by its name preceded by a "*" character and a space.

The "control data" section contains two lines, each of which contains two variables. The two variables on the first line are NPAR and NOBS, the number of parameters and observations respectively featured in this file. The next line contains the variables NUMIN and NUMOUT, the number of template and instruction files respectively featured in this file.

The "parameter data" section contains NPAR data lines. Each line contains two entries, the first being a parameter name, the second being the value of the parameter.

The "observation data" section contains NOBS data lines each containing one entry, this being the name of an observation.

The "model command line" section contains one data line, this being the command which DRIVER1 must use to run a model through a system call.

The "model input/output" section contains NUMIN+NUMOUT lines, each containing two entries. For the first NUMIN lines these entries are comprised of the name of a template file

followed by the name of the corresponding model input file. The last NUMOUT lines are comprised of the name of an instruction file followed by the name of a model output file which is read using that instruction file.

Execution of DRIVER1 is initiated by typing its name at the command line. The name of its input file, as well as that of an output file which it must write, should be supplied in response to its prompts. After it has issued these prompts and received the filenames which it requires, DRIVER1 writes some information pertaining to its input dataset to the screen as it processes, and then queries, this dataset using various MODEL_INPUT_OUTPUT_INTERFACE module subroutines. It then writes all cited model input files using corresponding template files and supplied parameter values. Next it runs the model. When the model has finished execution DRIVER1 reads all model output files, listing the values read therefrom to its own output file. This file can be inspected using a text editor.

A DRIVER1 input file named *driver1.dat* has been supplied. Template and instruction files cited in this file are also supplied. So too is a "model" named VES required for the running of this example. Source code is provided in file *ves.f*, while *ves.exe* is a PC executable version of this program. (VES computes apparent resistivities over a layered earth at multiple electrode separations.)

4.3 The DRIVER2 Utility

DRIVER2 USE's both the MODEL_INPUT_OUTPUT_INTERFACE and PARALLEL_RUN_MANAGER modules.

The operation of DRIVER2 is not unlike that of DRIVER1. However model runs are undertaken in parallel.

Execution of DRIVER2 is initiated by typing its name at the screen prompt. Immediately upon commencement of execution it prompts for the name of an input file. This file is identical to that of the DRIVER1 input file except for the fact that parameter values are not cited after parameter names in the "parameter data" section of this file; only parameter names are required.

Next DRIVER2 prompts for the name of a run management file. This file has the same format as that of the PEST run manager file. An example of such a file is provided below.

```
prf
2  0   0.2 0  0
Slave_1  .\test1\
Slave_2  .\test2\
2 2
```

A run management file used by the DRIVER2 utility.

The first line of the DRIVER2 run management file must contain the string "prf". The next line must contain 5 entries, all but the third being integers. The first entry is NSLAVE, the number of slaves involved in parallel run management. The next is the PEST IFLETYP variable (which is required to be zero when read by DRIVER1). The third is the strategic waiting time in seconds used in stabilising the parallel run management process. (This is multiplied by 100 for computation of the IWAIT variable used by the

PARALLEL_RUN_MANAGER module.) The integer PARLAM variable following that is ignored by the PARALLEL_RUN_MANAGER module. Then follows the REPEATRUN variable, which should be set to 1 or 0.

Then follow NSLAVE lines, each of which must possess two entries. The first is the name of a slave (ASLAVE) while the second is the name of the working directory (ASLDIR) for that slave. Following that is a line containing NSLAVE integer entries, these being expected model run times (in seconds) for all slaves.

After having read the run management file, DRIVER2 checks for the presence of all slaves. Once it has verified that at least one slave is alive and well, it prompts:-

```
How many run packages do you wish to implement?
```

Respond to this with an appropriate integer. For each run package, DRIVER2 then asks:-

```
Enter name of parameter value file for run package npackage:
```

where *npackage* in the above prompt is the package number. An example parameter value file is depicted below.

```
5
ro1   4.0     1.0    40.0   20.0    1.0
ro2   10.0    2.0     1.0   20.0    1.0e-2
ro3   1.0     1.0    20.0   20.0    1.0
h1    1.5     2.0     1.0   10.0    5.0
h2    10.0    2.0     1.0   10.0    2.0
```

Parameter value file used by DRIVER2.

The first line of the DRIVER 2 parameter value file contains the number of model runs required in this run package (i.e. NRUN). Then must follow a line of data for each parameter listed in the DRIVER2 global input file. Each of these lines must contain NRUN+1 entries. The first entry is the parameter name. Then, in each case, must follow NRUN values for that parameter. Note that parameter names in the parameter value file must be supplied in the same order as in the DRIVER2 main input file.

DRIVER2 then prompts:-

```
Enter name for observation output file for run package npackage:
```

in response to which a suitable output file should be named. DRIVER2 then supervises slaves as they carry out the requested model runs. When these runs are complete, DRIVER2 writes model outputs corresponding to the different parameter sets to the nominated output file.

The above process is repeated for each user-requested run package.

A DRIVER2 input file named *driver2.dat* is supplied. So too is a run management file named *driver2.rmf*. Three parameter value files named *parval_1.dat*, *parval_2.dat* and *parval_3.dat* are also supplied. DRIVER2 records run management details to the run management record file *driver2.rmr*. A list of completed runs is written to *model.runs*.

DRIVER2 also includes basic restart functionality. This is activated by starting it with the "/r" command-line switch. For restarting functionality to work, DRIVER2 must have been halted during implementation of its first run package during its previous execution. If, on its restarted run, it is provided with the name of the same parameter value file for its first run

package as that provided for its previous interrupted run, it will read the results of already-completed model runs from its restart file (named *driver2.res*) and then complete the remaining runs required of that package.