BeoPEST Programmer's Documentation

Willem A. Schreüder

willem@prinmath.com

Version 1.0: 15 November 2009

This is programmer's documentation for those interested into devling into the ugly depths of *BeoPEST* where jobs get scheduled and the master and slaves communicate. It is not required reading for using *BeoPEST*, but rather intended as internal documentation for those developing and maintaining *PEST*. And it is definately not for the feint of hart. Purists should write better code.

This is the initial draft, and it is pretty rough. As the code matures this documenation will hopefully improve to rise to the high standards set by the exquisite *PEST* documentation.

Please email any corrections, suggestions and improvements to the author. Improvements, bug fixes and clever ideas are always welcome.

Introduction

This documentation is intended to document the communications interface employed by *BeoPEST*. *BeoPEST* is a version of parallel *PEST* that that employs smart slaves and point-to-point communications to transfer data between the master and slaves.

BeoPEST comes in two flavors: *BeoPEST/TCP* and *BeoPEST/MPI*. Both flavors exist in the same code and both flavors can be activated at compile time. In fact, the *BeoPEST* executable is named ppest and by default will use the file based communications employed by ppest. However, by specifying the /M flag, the code will operate in *BeoPEST/MPI* mode, while specifying the /H flag the code will operate in *BeoPEST/TCP* mode.

To compile the code in *BeoPEST* mode, the -DBEO command line flag must be specified. To enable MPI at compile time the -DBEOMPI command line flag must be specified. Without the BEOMPI flag, *BeoPEST* will be compiled in TCP/IP mode only.

The source code for *BeoPEST* is typically saved in the src directory. The programs are then compiled in directories named for the architecture. The 64 bit linux code is compiled in x86_64-linux, the 32 bit linux code is compiled in i686-linux, the OSX code (which is both 32 and 64 bit) is comiled in x86_64-darwin, etc. The source code directory is referenced in the makefile using the VPATH command. This procedure is not required, but is convenient when compiling the same source code for multiple architectures.

Note that the makefile supplied with *BeoPEST* is different than the makefile distributed with *PEST*. It builds `pest`, `ppest` and the utilities in a single makefile.

The following sections describe the *BeoPEST*implementation and how it interfaces with *PEST*. The first section describes the high level interface to *BeoPEST*. This level of functionality allows both MPI and TCP/IP modes to operate using a single interface that hides the complexity of actually passing the messages. This level is appropriate for actually implementing communications between the master and slave nodes. The interface was designed to abstract the passing of messages sufficiently to hide the implementation differences between using MPI and TCP/IP.

The next section describes the top level interface. This interface is really simple and consists of basically two functions: RUNMASTER and RUNSLAVE. These functions are designed for interfacing with *PEST*. It is presented second as a demonstration of using the top level interface to run a set of runs on the slaves.

The third section describes the low level interface. The low level interface describes the TCP/IP functions that implement the actual sending and receiving of messages. The MPI low level interface is well documented elsewhere, so the low level interface described here only describes the TCP/IP functions.

High Level Interface

The *BeoPEST*high level interface consists of a series of FORTRAN routines. These routines implements the *BeoPEST* communications scheme independent of the low level implementation. The routines hide complexities such as whether TCP/IP or MPI is used as the transport mechanism.

```
MODULE BEOPEST
```

This module exposes a few variables that can be publically used. These variables should not be changed outside the *BeoPEST*routines. The variables made available in this way are:

LOGICAL BEOMASTER	true on the master node
LOGICAL BEOSLAVE	true on the slave node
LOGICAL BEOLASTLOC	true if the last run should be run locally
INTEGER MAXNODE	maximum number of *BeoPEST*slaves

A module BEOPRIV is used to store variables that *BeoPEST*considers to be private and only to be seen inside *BeoPEST*. These variables should NEVER be used. Should you find that a variable from there is useful, we should agonize about moving that variable to the BEOPEST module.

```
SUBROUTINE BEOINIT(HOST,MAXNODE)
CHARACTER*(*) HOST
INTEGER MAXNODE
```

This function is used to initialize *BeoPEST*. It must be called before any other BEO* function. The HOST and MAXNODE variables are named for their use in TCP/IP mode, and are complete misnomers in MPI mode. The HOST string is the name of the host specified by the user on the command line. This consiststs of the host:port or :port character string. BEOINIT will parse this string and set the BEOMASTER and BEOSLAVE logical variables appropriately, as well as open the TCP/IP connections. The MAXNODE parameter is used by BEOINIT to statically allocate memory for up to MAXNODE slaves. The amount of memory per slave is small, so this should be set generously, because it cannot be changed during the run. A value like 8192 or greater is appropriate.

MPI mode is selected by setting MAXNODE to less tahn or equal to zero. In this instance *BeoPEST*will use the MPI_COMM_SIZE call to determine the number of slaves available and set BEOMASTER to true for node 0 and BEOSLAVE to true for all other nodes. The HOST string is used to specify a directory. *BeoPEST*will append the node number to this string and change the working directory to this directory in BEOINIT.

BEOINIT performs a number of other functions like allocating memory for *BeoPEST*'s internal use. If the allocation fails, BEOINIT will not return. In TCP/IP model, BEOINIT will also open a TCP/IP connection to the master if it is a slave. If this connection fails, BEOINIT will not return but stop with an appropriate error message. *BeoPEST*will try hard to establish this connection, so BEOINIT may keep trying to establish this connection for seconds to minutes before failing, but will generally return in milliseconds.

```
SUBROUTINE BEOFIN
```

This function is used to shut down *BeoPEST*in an orderly fashion. On the MASTER, this function will send a special message to all the slaves causing them to shut downi before closing the connections, and then print run time statistics for all the slaves.

On the slaves *BeoPEST*will close the connection. This is particularly important in MPI mode because most MPI implementations will complain if not properly terminated.

```
SUBROUTINE BEOCAST(NPAR,NOBS,CSUM)
INTEGER NPAR
INTEGER NOBS
INTEGER CSUM
```

This function should be called once and only once shortly after calling BEOINIT and other BEO* functions. On the master, the number of parameters, number of observations and an integer checksum should passed to this function. As each slave connect, these values will then be sent to the slave.

On the slaves, the BEOCAST function will return the number of parameters, observations and the checksum when it connects to the master. The slave should check that it agrees with the values received from the master or stop immediately because when the master and slave disagrees on the number of parameters, observations or checksum, the slave is connecting to a master that is reading a different *PEST*control file. This sanity check should be done as soon as practical after reading the *PEST*control file. Note that this functionality could have been built into the BEOINIT function, but is provided separately so that BEOINIT can be called before even reading the *PEST*control file. It is, however, crucial that BEOCAST be called as it must allocate memory that depends on the number of parameters and observations that will be used for communicating with slaves.

```
SUBROUTINE BEONODE(KP0,KP1)
INTEGER KP0,KP1
```

This function is used to obtain the number of slaves available. In MPI mode, KP0 will always be 1 and KP1 will always be the number of slaves and will never change. However in TCP/IP mode, KP0 will be one or greater and KP1 will vary depending on the number of available slaves. These numbers can change at any time in TCPI/IP mode.

The recommended usage of BEONODE is to always call BEONODE to get indexes for the first and last node number before every loop that scans slaves, and then loop from KP0 to KP1. In MPI mode this introduces no overhead.

```
LOGICAL FUNCTION BEOGET(PAR)
DOUBLE PRECISION PAR(NPAR)
```

This function should be called ONLY on a slave. The BEOGET function does a blocking read and returns NPAR double precision values which is a set of parameters that the master wants the slave to run.

The function blocks and does not return until a set of parameters are received. The funtion returns true if a set of parameters are received from the master and the values are packed in the PAR array passed to BEOGET. The function will return false if the master sends the done message. The slave should react to a BEOGET false by calling BEOFIN to shut down the connection as the master will be sending no more parameters.

```
SUBROUTINE BEOPUT(PAR,OBS)
DOUBLE PRECISION PAR(NPAR),OBS(NOBS)
```

This routine should ONLY be called from the slave to return a set of parameters and observations to the master. The PAR array should contain the parameters as written and the observations as read from the model output.

This function should return quickly as the master should have performed a non-blocking read to rceive these values. Since *BeoPEST*uses reliable communications, this call will only fail if the master crashed. Therefore a write error in this function will cause the slave to also crash in response to the crash on the master.

```
SUBROUTINE BEOPUSH(NODE,PAR)
INTEGER NODE
DOUBLE PRECISION PAR(NPAR)
```

This routine should ONLY be called on the master. The NODE value should be the node number that is the node on which the master wants the run to be done. The parameters to be passed to the slave should be in PAR.

This function will return quickly and never fail. If the slave has died, it will be detected later by BEOSTAT. The routine will allocate memory and post a nonblocking receive that will receive the results from the slave. The BEOSTAT function should be used to test when the slave is done.

```
INTEGER FUNCTION BEOSTAT(NODE)
```

The BEOSTAT function should ONLY be called on the master. It is used to check the status of the slave. The NODE value is the node to check. The BEONODE function should be used to determine wha nodes can be checked.

The function returns 0 if the run requested with BEOPUSH is still working. This means that the results has not been received yet and BEOSTAT should be called again to check if the run finished.

The function returns 1 if the slave is idle. This means could mean that the slave has finished running the job. It is important that if the the slave finished running a job, the results of that run be processed before scheduling the next job to avoid destroying the returned results. The results should be stored using the BEOSTORE function.

If the slave died, the BEOSTAT function will return -1. If a job had been scheduled on that slave, the results are irretrievably lost and the run should be rescheduled.

```
SUBROUTINE BEOSTORE(NODE,JOB,PARREG,OBSREG)
INTEGER NODE,JOB,PARREG,OBSREG
```

This function is used to store the results returned by a slave to the file register. BEO-STORE should be called after BEOSTAT returns 1 to indicate that the slave has completed the scheduled job. NODE is the node that returned the value, and JOB is the job number which is used to select the register to which results are saved. PARREG and OBSREG are the logical units needed by the STORE_PARALLEL_REGISTER as logical units.

```
DOUBLE PRECISION FUNCTION BEOWALL(NODE)
INTEGER NODE
```

This function returns the wall time it took to execute the last run on this node.

```
SUBROUTINE BEOFATAL(MESSAGE)
CHARACTER*(*) MESSAGE
```

This function will abruptly terminate the process with the node number and the supplied message. This function should be called if there is an unrecoverable error. It is best suited to being called on the slave. On the master, much more care should be taken than such a harsh action.

```
DOUBLE PRECISION WALLTIME()
```

This function returns the number of seconds since 1-Jan-1970. This is the UNIX time index. Values returned can a best be accurate to a millisecond.

```
CHARACTER*256 FUNCTION BEOTEXT(NODE)
INTEGER NODE
```

This function returns the identifying text string that descripes the node. In MPI mode, this is the node number which is an integer between 0 and the number of nodes. In TCP/IP mode, this is a text string that may include the host name, architecture and similar information sent to the master by the slave. This is an untrusted string and should only be used to print identifying information.

Top Level Interface

*BeoPEST*is initialized in the *PEST* PARSE_COMMAND_LINE function. If neither the /H or /M flags are specified, both the BEOMSATER and BEOSLAVE logical variables are set to false. However, if etiher the /H or /M flag is specified, then BEOINIT is called. With the /H flag, the TCP/IP approach is selected by specifying 8192 as the maximum number of slaves. This number is set at compile time and should be increased if that number of slaves becomes commonplace. In MPI mode selected by the /M flag, the number of nodes are set to 0. The user supplied text string interepreted to be the HOST:PORT or directory string in TCP/IP and MPI modes, respectively, is passed verbatim.

The BEOMASTER and BEOSLAVE variables should be used to dynamically determine whether *PEST*should operate in master, slave or no *BeoPEST*mode.

If BEOSLAVE is true, the main routine is short circuited by calling RUNSLAVE right after READ_PEST_DATA. Therefore in slave mode, the main program will rad the *PEST*control file which also in turn reads the template and instruction files, and then call RUNSLAVE. Upon returning from RUNSLAVE, the slave simply shuts down by calling PEST_FILES_CLOSE, PEST_DATA_DEALLOCATE and BEOFIN.

If BEOMASTER is true, the *PEST*functionality is invoked as usual. However, the BEOMASTER variable is tested in a few places to short circuit, for example, reading the rmf file.

The main switch occurs in the RUN_PEST function. In order to schedule the runs on the slave, the RUNMASTER function is used to execute the runs. This function is equivalent to the DORUNS routines used before. Runs to do is passed to RUNMASTER using the PARREG register file. Similarly, the returned results are returned using the OBSREG register file.

```
SUBROUTINE RUNMASTER(PARREG,OBSREG,NJOB,JFAIL)
INTEGER PARREG,OBSREG,NJOB,JFAIL
```

This routine runs a set of jobs. The number of jobs are determined by the NJOB variable. The result of this function is returned via the JFAIL variable.

The parameters to run are passed using the *PEST* register file mechanism. After running the jobs, the actual parameters ran by the slave are saved in the same parameter register file, while the observations are saved in the observation register file. The logical units for the parameter and observation register files are passed in the PARREG and OBSREG variables.

The RUNMASTER function tracks the jobs using MODE variable. This array is dynamically allocated and dimensioned to NJOB. It is initialized to -1 indicating that no jobs have been run. After a job was successfully run, the MODE variable for the corresponding job is set to 0. When the job is schedule on a node, the MODE variable is set to the slave node. This is not actually used, but is potentially useful.

The STAT variable is used to track active nodes. It is dimensioned to MAXNODE which is the number of MPI nodes, or the maximum nomber of nodes set in TCP/IP mode. It is initially set to -1 marking all nodes as offline. The BEOSTAT routine is used to test all nodes in the range specified by BEONODE. A STAT value of zero is used to indicate that the slave is idle, while a positive value is used to indicate which job is scheduled on that slave.

When BEOSTAT indicates that a node is idle, the current value of STAT is checked. If STAT is positive, this means that a job previously scheduled on this node has completed. The MODE variable for that job is therefore set to zero indicating that the job has been processed, and the BEOSTORE routine called to store the returned values in the register files. The STAT variable is then set to zero to indicate that the slave is idle.

If BEOSTAT indicates that the node is idle, a previously dead slave indicated by a negative STAT value is marked as idle by setting STAT to zero.

If BEOSTAT indicates that a node has died, the value of STAT is checked to see if a job had been scheduled on that node. If STAT is positive, that job failed. The MODE variable for that job is therefore set to -1 (not scheduled). Regardless of the previous value of STAT, a dead slave will cause the value of STAT to be set to -1.

After checking the status of all the nodes with BEOSTAT, RUNMASTER will attempt to schedule any unscheduled jobs. The idle variable is used to store the fastest free slave. This is done by checking all nodes with STAT set to zero. The BEOWALL function returns the time used to run the last job. If idle is zero, no idle slave was found.

The next variable is used to store the number of the first unscheduled job found in MODE. If an unscheduled jobs exists and a slave is idle, the BEOPUSH routine is used to schedule that job.

If there no job completed or was scheduled, the master will sleep for 5 milliseconds before trying again.

```
SUBROUTINE RUNSLAVE
```

The RUNSLAVE routine encapsulates all of the smart slave activity. The routine only requires that BEOINIT be called prior to calling RUNSLAVE to initialize the communications infrastructure.

The RUNSLAVE routine calls the standard *PEST* routines IOCTL and READINS to read the template and instruction files. It then enters an endless loop calling BEOGET to read parameters to run.

When a set of parameters are received from the master, the standard *PEST* INWRIT routine is used to write the model input files. The SYSTEM call is called next to run the model. Next the standard *PEST* OUTRD routine is used to read the model results. Finally BEOPUT is used to return the parameter and observations to the master.

```
SUBROUTINE FASTSLAVES(NUMSLAVE)
INTEGER NUMSLAVE
```

This function returns the number of fast slaves. Fast is defined as an execution time no more than twice the execution time of the fastest slave.

MPI Low Level Interface

The MPI low level interface is implemented using point-to-point MPI messages. Communications are initiated in the BEOINIT routine, which also gets the total number of nodes. Node 0 assumes the role of master, while all other nodes will act as slaves.

In MPI mode, the master and slaves individually read the *PEST* control file. In the BEOCAST routine, an MPI_BCAST call is used to ensure that the master and slaves agree on the number of parameters, number of observations and a checksum. On the master, the BEOCAST routine also allocates buffers that will be used to receive the parameters and observations from the slaves. This occurs in the BEOCAST routine because this is the earliest call where the number of parameters, number of observatins and number of slaves are known.

The BEOPUSH function on the master schedules a run on a slave. The BEOPUSH function first posts two nonblocking receives, one for the parameters and one for the observations. The buffers allocated in the BEOCAST call are used to receive these values. The parameters and observations are distinguished by using different tags. After posting the receives, master sends the number of parameters and the array of parameters as blocking sends. The number of parameters is used as a signal, with a parameter count of zero indicating the end of the optimization.

The slaves executes a blocking read using MPI_RECV in the BEOGET function. It first receives the integer parameter count, and then the actual parameters as double precision. After running the model and obtaining the results, the parameters and observations are returned using a blocking MPI_SEND. Tags are used to distinguish the parameters and observations.

The BEOSTAT function is used on the master to test for completion of the slave. The MPI_TEST function is used in conjunction with the pointer returned by the MPI_IRECV to test whether both the parameter and observatins were returned.

TCP/IP Low Level Interface

The TCP/IP low level interface is written in C. This choice of language is to a large extent dictated by the fact that the TCP/IP implementation is a socket based communication which was originally implemented on the UNIX system which was written in C. In fact, C was largely invented in order to write the original UNIX operating system. Therefore the C language is the natural choice in which to implement the TCP/IP communications. It is also worth nothing that MPI is also written in C. However, both the MPI and TCP/IP implementations in C have bindings to FORTRAN to allow it to be used with the high level *BeoPEST* interface. The interface is therefore described in terms of FORTRAN here.

It is very important to note that the TCP/IP implementation is NOT intended to be a full fledged message library the way MPI is. There are many implementation specific assumptions that makes this implementation unsuitable for general purpose use. For example, it is assumed that point to point communications will always be from the master to the slaves. Slaves never communicate with each other. MPI allows slave-to-slave communications in general, but this requires a far more complex implementation. This code takes LOTS of ugly shortcuts in order that greatly simplify the implementation. Therfore if you want to change the implementation it would wise to make sure that it does not violate any of these assumptions.

```
SUBROUTINE TCPOPEN(MAXSLAVES,NAME,NODE)
INTEGER MAXSLAVES
CHARACTER(*) NAME
INTEGER NODE
```

This function establishes the connection between master and slave nodes. The NAME variable is used to identify whether the function should actin in master or slave mode. If the NAME is :d where d is an integer port number, the function acts in master mode and opens that port to listen on and accept connections from slaves. If the NAME is h:d where h is a hostname and d is an integer port number, the function acts in slave mode and connects the master at the specified host and port.

The NAME variable is unusual in that it must be a C style string, that is terminated by a null.

In master mode, the MAXSLAVES variable is used to allocate memory for that many connections. Only about 64 bytes are allocated for each potential slave in this function, so setting MAXSLAVES much larger than then number of possible slaves is not a penalty. Currently this is set to 8192 in *BeoPEST*.

In slave mode, this function will establish a TCP/IP connection to the master. If a connection cannot be established, either due to the master having died or a communications faulure or some similar reason, the function will print an error message and exit with a fatal error. If the connection is established, the slave will receive a two byte integer from the master that will establish whether the master and slave has the same byte gender. If the slave detects that it has the opposite byte gender than the master, the slave will remember to reverse the byte gender on all its communications with the master. Finally the slave will send a string identifying itself to the master which may be useful for identifying individual slaves.

On completion the TCPOPEN function will set the NODE variable. For the master this will be zero. For a slave this will be a positive integer which is the node number assigned by the master.

```
SUBROUTINE TCPSEND(NODE,BUFFER,LEN,NUM)
```

```
INTEGER NODE
* BUFFER INTEGER LEN
INTEGER NUM
```

The TCPSEND function is used to send BUFFER to the node number NODE. This can be called on the master to send to any of the slaves. On the slaves it can only be used to send to the master. BUFFER can be any variable and is passed as a pointer to this function. LEN identifies the size of the items passed in BUFFER. This would typically be 4 for integers and 8 for double precision floating point numbers. NUM is the number of these values to send.

The number of bytes transferred by TCPSEND is LEN*NUM. However, in those instances where the byte gender of the master and slave is different, every LEN bytes will be reversed on the slave to adjust the byte gender to match the master.

TCPSEND performs a blocking send. The function will not return until the operating system took responsibility for all the bytes to send. This does not always mean that it has been received at the other end. If the send fails, an informational message will be printed, but failed sends will return without an error message and rely on future calls to flag the fatal error. Since TCP/IP is a reliable protocol, a failure means that the communications infrastructure has failed or the process being sent to has died. This will cause the connection to be dropped and result in a future failure.

```
SUBROUTINE TCPRECV(NODE,BUFFER,LEN,NUM)
INTEGER NODE
* BUFFER INTEGER LEN
INTEGER NUM
```

This function is used to perform a blocking read. While it potentially can be used on the master, this is not used because the master should never sit and wait for a client. Instead, the master should use TCPPOST which is a nonblocking read. Slaves, on the other hand, can use TCPRECV to wait until the master gives them a task. This consumes no effort on the slaves.

The NODE selects the node from which to receive the input. This will always be zero (the master) for the slaves. The BUFFER is where the result should be placed, which is an arbitrary variable passed as a pointer. How many bytes the slave expects to receive is set by LEN and NUM. LEN specifies the length of the variable type, typically 4 for integers and 8 for double precision reals. NUM sets the number of values to receive. The total number of bytes received is LEN*NUM. Both LEN and NUM must be specified because if the master and slave have differing byte gender, the slave must reverse the order of the bytes withing each variable.

```
SUBROUTINE TCPPOST(NODE,TAG,LEN,NUM)
INTEGER NODE
INTEGER TAG
INTEGER LEN
INTEGER NUM
```

The TCPPOST subroutine is used to post a nonblocking read. It should ONLY be used on the master. The NODE is the slave from which the answer is expected. The TAG is used to allow multiple receives to be posted simultaneously. Specifically, a TAG of 0 and

1 allows both the parameters and observations to be received. In actuality, this does not truly implement tagged messages as supported by MPI. The concept of a tag in MPI allows multiple messages to be sent from one process to another, and the variable type and tag can be used to separate the different messages. Such sophistication is not required here, so this this implementation only does lip service to the concept. Specifically, the TCPPOST only allows the parameters and observations to be received as two seperated buffers instead of a single buffer. This is done as two TCPSEND calls on the slave. The slave does not tag the sent messages. Only the order of the messages (which is guaranteed to arrive in order) separates the messages which TCPPOST will separate into the tags 0 and 1.

It is not required that TCPPOST calls tag 0 before tag 1. However, the first send from the slave will go to tag 0 and the second to tag 1.

Like TCPSEND and TCPRECV the LEN and NUM values set the length of the variables (4 for integers and 8 for double precision reals) and the number of variables. Since TCPPOST is ONLY used on the master, only LEN*NUM is required because the byte gender adjustment is done on the slave. Therefore the requirement that both LEN and NUM be specified is simply to provide a consistent interface.

```
SUBROUTINE TCPTEST(NODE,FLAG)
INTEGER NODE
INTEGER FLAG
```

This function determines if the nonblocking receives posted by TCPPOST has completed. It is assumed that two TCPPOST calls are always done (TAG 0 and 1) so TCPTEST will test whether both receives have completed.

TCPTEST should ONLY be called from the master. The NODE specifies which slave to test. The FLAG variable returns the status of the transfer. A positive value indicates that the receive has completed and therefore that sets of values from the slave is available. A value of 0 indicates that the messages have not yet arrived, either because the slave is still working and have not sent the results yet, or only some of the bytes have arrived yet.

Since TCP/IP is a reliable protocol, an error resulting from a failed slave process or a failure in the communications infrastracture can be reliably detected. If such a failure is detected, a value of -1 is returned in FLAG. The master should act on this as a failed slave and reschedule the job.

When the TCPTEST returns 1, the TCPLOAD function can be used to access the received messages. It is important that this occurs before doing the next TCPPOST to prevent the buffers being overwritten.

```
SUBROUTINE TCPLOAD(NODE,TAG,BUFFER,LEN,NUM)
INTEGER NODE
INTEGER TAG
* BUFFER
INTEGER LEN
INTEGER NUM
```

The TCPLOAD function is used ONLY on the master to retrieve the messages received from the slaves. The NODE and TAG is used to identify the specific message and the LEN and NUM is used to determine its size. LEN and NUM is actually redundant since it is know how large the message is from the TCPPOST call, but it is retained here for the sake of paranoia.

The function copies the bytes from an internal buffer into the user supplied BUFFER. Note that TCPLOAD must only be called after TCPTEST indicates that the message has completed.

```
SUBROUTINE TCPCLOSE()
```

The TCPCLOSE function closes all the TCP/IP connection. On the master multiple connections are closed, while on the slave only the one connection exists and is closed. This function does nothing other than close the connections gracefully and free the associated memory.

```
SUBROUTINE TCPCAST(NPAR,NOBJ,CSUM)
INTEGER NPAR,NOBJ,CSUM
```

The TCPCAST function is used to perform an asymmetric broadcast of the number of parameters and observations and a checksum. This is necessary because unlike MPI, slaves can joint the master at any time. Therfore the master cannot wait for all the slaves to join before doing the broadcast.

TCPCAST must ONLY be called on the master. This sets the number of parameters and objects and the checksum on the master and returns immediately.

When a slave connects to the master, the master will send these parameters to the slave. The slave must do a TCPRECV to receive these values and check that the master and slave agrees on the number of parameters and observations and the checksum.

```
SUBROUTINE TCPNODE(LIVE,WALL,NAME,NUM)
INTEGER LIVE()
DOUBLE WALL()
CHARACTER*(*) NAME()
INTEGER NUM
```

This function is used to check for new slaves. The function should be called frequently since this is where the connections from slaves are accepted. If the master fails to call this function for an extended period of time, newly connection slaves may give up thinking that the master process has died because the connection was not accepted.

The NUM variable is set to the highest node number among the active slaves. The LIVE array is used to indicate which nodes are actually live. When a new slaves connects, the corresponding entry in the LIVE array is set to -1 to indicate that the slave is idle. The LIVE variable is used internally by the BeoPEST routines to keep track of slaves.

The new slave sends to the master a speed estimate and an identifying string. The speed estimate is used to initialize WALL which is the execution time of a job and NAME which is an identifying string for the slave. By setting WALL to zero the slave will be scheduled a job as soon as possible.

Coming Attractions

One of the most difficult problems for *BeoPEST* is load balancing. Since some sets of parameters may cause the model to run very slowly while others run fast, it is difficult to keep all the slaves busy doing useful work. This problem is exacerbated when some of the slaves are older machines that are not as fast as others. It is difficult for *BeoPEST* to detect whether the problem is that the set of parameters just makes the problem hard to solve, or whether the hardware is simply slow or busy doing something else.

A simple solution to this problem relies on the fact that the model runs are idempotent. This simply means that you can run the job several times without negative side effects. The concept is therefore that when there are idle slaves and some jobs have been running for a while but not yet completed, the master can reschedule those jobs on the idle slaves in the hope that they can get the job done faster.

In order to implement this, all that is required is that the RUNMASTER program be adapted to perform this over scheduling. Then when the first of the solutions are obtained, the master must notify the slave that it can give up on the job. This is a bit tricky. The current RUNSLAVE is purely sequential. It performs a blocking read for the parameters, runs the model and returns the results. In order to receive notification that the result is no longer needed, the slave must have the ability to receive the *never mind* message, abort the running process with prejudice, and block for the next scheduled job. On the master it is necessary to clean out the posted receives.

This task is nontrivial to implement in a portable way. On a UNIX/Linux/OSX system, the fork() mechanism can be used to spawn a child process. The slave can then use the select() mechanism to monitor for any incoming messages for the master without blocking. If the *never mind* message is received form the master, the slave can use the kill() mechanism to terminate the child process. Alternatively if the child process terminates, the slave will be notified via a signal and can proceed with processing the results.

A similar mechanisms is available on Windows systems, but is quite different in details from the UNIX/Linux/OSX framework in implementation. Therefore implementing this feature needs to be done with great care. Since this delves fairly deep into the operating system specifics, this functionality will have to be implemented in C just like the TCP/IP functionality.