Management of the errors in parallel MPI

Summary:

One provides in this document the details of implementation concerning the management of the errors during parallel executions MPI.

1 Operation

The parallel operation on different processes (using MPI) requires a particular treatment in the event of error.
Indeed, if nothing is done, if an error does not occur on all the processors at the same time, i.e. between the two same communications, a processor stops and the others indefinitely expect it the following communication until stop CPU and loss of all calculation.

This particular treatment consists in checking before engaging a total communication that all the processors are with go and in which state they are (they transmitted an error message or not).

- If they all are to go and that none met an error, one continues while proceeding to the communication envisaged.
- If they all are to go but that at least a processor transmitted an error message, one asks all the processors to stop as usual (while raising an exception).

The behavior is then the same one as into sequential: error <S> (exception) and thus safeguard of the files of the base.

- So at least one of the processors is not with go (within an agreed time), it is that this processor is blocked on a task much longer than on the other processors, or in an infinite loop, a programming error, or it left brutally.

In this case, one is obliged to stop the execution of the remaining processors brutally. The base should not be saved.

Moreover, one makes a communication in `END` to recover the number of alarms emitted (and not been unaware of) by each processor. For the diagnosis, made on the processor #0, one emits an alarm which gives simply the number of alarms emitted by processor.
This avoids “to flunk” an alarm which would have occurred only on one processor.

2 Details of implementation

Notice

This paragraph consists of notes of development and should make it possible in an external eye to understand how that was done.

2.1 Total state of the execution

It is necessary to store the state of the processors to try to stop a maximum of calculations properly.

One must store: ok/error, to separate proc #0/autres

Functions necessary:
- to say that all is ok everywhere.
- to say that an error was seen on proc #0 or others.
- to know if all is ok.
- to know if error about proc #0 or others.

The state is stored in one `COMMON` and two routines exist to question (`GTSTAT`, for *get status*) and to affect the state (`STSTAT`, for *set status*). One uses constants to simplify the reading (see `aster_constant.h`).

Contents of `aster_constant.h` :

```
#define ST_OK       0
```

```
#define ST_AL_PR0  1    /* alarm one processor #0 *
#define ST_AL_OTH  2    /* alarm one another processor *
#define ST_ER_PR0  4    /* error one processor #0 *
#define ST_ER_OTH  8    /* error one another processor *
#define ST_UN_OTH 16    /* undefined status for another processor *
```

One uses logical operations bit bit to store and know if a state is positioned.

2.2 Communications not blocking

The key to detect that certain processors do not answer go is to use communications MPI not blocking.
The starting point of the specific treatment is in u2mesg during the emission of the error message.

2.2.1 u2mesg

In the event of error, one prevents the proc #0 while calling mpicmw (). Idem in Utmess.py, while calling, aster_core.mpi_warn ().

2.2.2 mpisst

Sending with the proc #0 ST_OK or ST_ER (MPI_ISEND tag CHK, nonblocking send) and expects the answer of proc #0 (MPI_IRECV tag CNT, nonblocking receive). One puts a time-out not to wait indefinitely. If the proc #0 does not answer within the time, one calls MPI_Abort via mpistp (1).

If one sent ST_OK, one wants to know if one must continue or not.

If one sent ST_ER, right knowledge is wanted if the proc #0 answers (in this case one stops properly), if not one must stop the execution.

If not time-out, one turns over the answer of proc #0: ST_OK (all is well), ST_ER (to make a clean stop).

2.2.3 mpicmw

To alert the proc #0 that a problem was encountered

- On proc! = 0, one position ST_ER_OTH (error on a processor other than #0) and one sends ST_ER with the proc #0 with mpisst (ST_ER).

The answer of proc #0 is ST_ER, one continues as into sequential (exception, closing of the bases or abort).

- On proc #0, one positions ST_ER_PR0 (error specific to the proc #0) and one calls mpichk ().

2.2.4 mpichk

Called before making a total communication to check that all is well and if not to act consequently.

- On proc! = 0, one send ST_OK with the proc #0 with mpisst (ST_OK) and one expects the answer of the proc #0 to know if one must continue or stop.

  If the proc #0 answers that the execution should be stopped, one calls mpistp (2).

- On proc #0, one expects the answer of all the other processors (MPI_IRECV tag CHK nonblocking receive) + a time-out not to wait indefinitely.

- If a proc met an error (and thus sent ST_ER), message “error on the proc #i” + STSTAT (ST_ER_OTH).

- If one of the procs does not answer within the time, message “the proc #0 waited too much” and error 'E' “the processor #i did not answer” + STSTAT (ST_UN_OTH).

- To the processors present with go, one answers ‘to continue’ or ‘to stop’ (MPI_SEND tag CNT, blocking send). In the event of error on the proc #0, one sends to stop. To stop, one invites mpistp (2).

- If one of the processors were not with go, the proc #0 stops the execution with MPI_Abort : one calls mpistp (1).

mpichk provides a code return: 0 = ok, 1 = nook.

2.2.5 mpistp

Used to stop the execution.

- mpistp (2): all the processors communicated their state, one can thus stop the execution properly with u2mess ('Me, 'APPELMPI_95'). If an exception were already raised by u2mess ('F' or ') precedent, it is necessary to avoid the recursivity and not to raise another exception. If no error were already emitted, the behavior is that of an error 'F' ordinary.

- mpistp (1): at least a processor did not answer (perhaps the proc #0), one must stop everyone including this one which does not answer. One is emitted u2mess ('Of, 'APPELMPI_99') who prints the message with 'F' (for the diagnosis) but does not emit exception -- what would cause a disconnection, and the continuation would thus not be carried out -- then one calls JEFINI ('ERROR') to start MPI_Abort.

- if ERREUR_F=' ABORT', mpistp (2) becomes mpistp (1).

- One should not carry out instruction after a call to mpistp (2), to make GOTO end of routine when one calls mpichk ().

2.2.6 mpicm1/mpicm2

Before beginning a communication, one calls mpichk () to check that there no was problem. To take account of the code return and to stop without making the communication!

2.2.7 jefini/MPI_Abort

Instead of stopping with ABORT (), one calls ASABRT (6) (6 corresponds to SIGABRT) who calls MPI_Abort.

It is essential to call MPI_Abort to be able to stop everyone, including the blocked processors. However MPI_Abort imply the end of the script launched by mpirun and thus the copy of the results of the repertoire of the proc #0 towards the total repertoire will not be able to take place (finally, that can depend on implementation MPI).
Thus “error in MPI” must involve “pas de bases saved” and in the event of error, the diagnosis is likely not to be very detailed (according to the implementation MPI, the files fort.8/fort.9 are or are not recopied in the repertoire of total work). The diagnosis is likely to be <F>_ABNORMAL_ABORT instead of <F>_ERROR.

2.2.8 Additional notes, precautions

***MPI_Abort* the execution did not stop.**

In MPI, it is necessary that the processors pass all by MPI_Finalize before leaving.
However in the interpreter Python, one leaves by 'sys.exit ()'which probably calls the function system'exit'and thus one cannot add call to MPI_Finalize before leaving. This is why, one records a function which carries out MPI_Finalize via 'atexit'.
The problem is that this function is also called after one MPI_Abort. The execution is thus blocked without stopping all the processors. A function is thus defined ASABRT who makes 'abort'or'MPI_Abort'in parallel and which positions a flag not to pass by MPI_Finalize in the function 'terminate'(cf "aster_error.c/h").

Precaution for calls FORTRAN since C

Since one calls routines FORTRAN since C, knowing that almost all the routines are likely to emit u2mess and thus of raising exceptions, it is imperative that the extension C of the module (aster or aster_core) one envisages try/except (in C) to treat this exception (and to turn over NO ONE in the event of error).
Indeed, the exception causes a disconnection of the execution. If there is not try, one is likely not to be replugged where one believes. A programming error would alert if none try was not set up more high.

Example:

```
          static PyObject* aster_mpi_warn (PyObject *self, PyObject
*arguments)
     {
         try {
             CALL_MPICMW ();
         }
         exceptAll {
             raiseException ();
         }
         endTry ();
         Py_INCREF (Py_None);
         return Py_None;
     }
```

2.3 Values of the deadlines

They is the times granted to the latecomers at the time of the communications not blocking.

Difference between two processors:

```
#0 ========|t0|...... |Ti|…

#i ==================|Ti|…
```

$[ti-t0]$: time granted by #0 to the processors #i. Thus if #1 arrives before #0, it must grant the same time to him: $t0-t1=ti-t0$.

The extreme case is:

```
#0 ==================|t0|~~~~~~|Ti|.v=========
                       ^          ^   v
#1 ========|T1|....... ^~~~~~~~~~^~~~v=========
                                   ^
#i ============================|Ti|||tf|======
```

* $t1$: arrival of the first processor #1

* $t0$: arrived from #0, #0 receives CHK of #1

* $t0+dt$: #1 expects the answer of #0

* ti : arrived from #i, #0 receives CHK of #i, #0 sends CNT with #1 and #i

* $ti+dt=tf$: #1 and #i receive CNT of #0

It is necessary thus that $tf-t0>ti-t0$. One limits the time of reception of the answer of #0 to $1.2\times[ti-t0]$

The value of time-out is fixed at 20% of remaining time CPU.