Management of the errors in parallel MPI

Summarized:

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

1 Operation

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

This particular processing 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 saves 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 FIN 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

Notices

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 computations 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 a COMMON and two routines exist to question (GTSTAT, for *get status*) and to affect 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. Communications

2.2 not blocking the word

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

2.2.1 In the event of

error, one prevents the proc #0 by calling mpicmw (). Idem in Utmess.py , while calling, aster_core.mpi _warn (). mpisst Sending

2.2.2 with the proc

#0 ST_OK or ST _ER (MPI_ ISEND tag CHK, nonblocking send) and expects the response 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 ST_OK

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

were sent, 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 response of proc #0: ST_OK (all is well), ST_ER (to make a clean stop). mpicmw To alert

2.2.3

the proc #0 that one encountered a problem On proc! =

•0, one position ST_ER_OTH (error on a processor other than #0) and one sends ST_ER to the proc #0 with mpisst (ST_ER). The response

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 (). mpichk Called

2.2.4 before

making a total communication to check that all is well and if not to act consequently. On proc! =

•0, one send ST_OK to the proc #0 with mpisst (ST_OK) and one expects the response 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 response of all other processors (MPI_IRECV receive tag CHK nonblocking) + 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: mpistp is called (1). mpichk provides

a return code: 0 = ok, 1 = nook. mpistp Used

2.2.5

to stop the execution. mpistp (2):

•all the processors communicated their state, one can thus stop the execution with u2mess properly ("Me, "APPELMPI_95"). If an exception were already raised by u2mess ("F" or ") preceding , it is necessary to avoid recursion and not to raise another exception. If no error were already emitted, the behavior is that of an ordinary error "F" . 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 emits a u2mess (", "APPELMPI_99") which prints the message with "F" (for the diagnosis) but does not emit exception -- what would cause a bifurcation, and the continuation would thus not be carried out -- then one invites JEFINI ("ERREUR ") to start MPI_Abort. so ERROR_

•F=' ABORT ', mpistp (2) becomes mpistp (1). One should not

•2) carry out instruction after a call to mpistp (, to make fine GOTO of routine when mpichk is called (). mpicm 1/mpicm

2.2.6 2 Before beginning

a communication, one invites mpichk () to check that there no was problem. To take account of the return code and to stop without making the communication! jefini/MPI

2.2.7 _Abort Instead of stopping

with ABORT (), ASABRT (6) is called (6 corresponds to SIGABRT) which calls MPI_Abort. It is essential

to call MPI_Abort to be able to stop everyone, including the blocked processors. However MPI_Abort implies the end of the script launched by mpirun and thus the copy of the results of the directory of the proc #0 towards the total directory will not be able to take place (finally, that can depend on implementation MPI). Thus "error
in MPI" must involve "basic not 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 total working directory). The diagnosis is likely to be <F>_ABNORMAL _ABORT instead of <F>_ERROR. Additional notes

2.2.8 , MPI_Abort precautions

***did not stop* the execution. In MPI, it**

is necessary that the processors pass all by MPI_Finalize before leaving . However in the interpreter
Python, one thus leaves by “sys.exit ()” which probably calls the function system “exit” and 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 MPI_Abort. The execution is thus blocked without stopping all the processors. One thus defines a function ASABRT which 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 modulus (aster or aster_core) envisages a try/except (into C) to treat this exception (and to turn over NO ONE in the event of error) . Indeed, the exception
causes a bifurcation of the execution. If there is no try, one is likely not to be replugged where one believes. A programming error would alert if no try were set up higher. Example: static

PyObject*

```
          aster_mpi_warn (PyObject *self, PyObject *arguments) {try
{CALL_MPICMW

         (); }
             exceptAll {raiseException

         ();
             } endTry (); Py_INCREF

         (Py_None
         ); return Py_None; }
         Values of the deadlines
```

2.3 It during acts of

the times granted to the latecomers the communications not blocking. Difference between two

processors: #0 ========|t0|.....

```
|Ti|… #i ==============

====|Ti|…: time granted
```

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

: #0 ==========

```
====|t0|~~~~~~|Ti|.v========= ^ ^ v #1 ======
                        ==|          T1|   .......
^~~~~~~~~~^~~~v========= ^ #i ============
                                    ==
==============|Ti|||tf|====== *: arrival of
```

the first $t1$ processor #1 *: arrived from

#0, $t0$ #0 receives CHK of #1 *: #1 `expects`

the response $t0+dt$ of #0 *: arrived from

#i, ti #0 receives CHK of #i, #0 sends CNT to #1 and #i * : #1 and #i receive

$ti+dt=tf$ CNT of #0 It is necessary `thus` that

. One limits the time $tf-t0>ti-t0$ of reception of the response of #0 to the value of time-out $1.2\times[ti-t0]$

is fixed at 20% of remaining time CPU.