Gestion des erreurs en parallèle MPI

Résumé :

On fournit dans ce document les détails d'implémentation concernant la gestion des erreurs lors des exécutions MPI parallèles.

1 Fonctionnement

Le fonctionnement en parallèle sur des processus différents (utilisant MPI) nécessite un traitement particulier en cas d'erreur.
En effet, si rien n'est fait, si une erreur ne se produit pas sur tous les processeurs au même moment, c'est à dire entre les deux mêmes communications, un processeur s'arrête et les autres l'attendent à la communication suivante indéfiniment jusqu'à l'arrêt CPU et perte de tout le calcul.

Ce traitement particulier consiste à vérifier avant d'engager une communication globale que tous les processeurs sont au rendez-vous et dans quel état ils sont (ont-ils émis un message d'erreur ou pas).

- S'ils sont tous au rendez-vous et qu'aucun n'a rencontré une erreur, on continue en procédant à la communication prévue.
- S'ils sont tous au rendez-vous mais qu'au moins un processeur a émis un message d'erreur, on demande à tous les processeurs de s'arrêter comme d'habitude (en levant une exception).

Le comportement est alors le même qu'en séquentiel : erreur <S> (exception) et donc sauvegarde des fichiers de la base.

- Si au moins un des processeurs n'est pas au rendez-vous (dans un délai convenu), c'est que ce processeur est bloqué sur une tâche beaucoup plus longue que sur les autres processeurs, ou dans une boucle infinie, une erreur de programmation, ou encore il a quitté brutalement.

Dans ce cas, on est obligé d'interrompre brutalement l'exécution des processeurs restants. La base ne doit pas être sauvegardée.

De plus, on fait une communication dans `FIN` pour récupérer le nombre d'alarmes émises (et non ignorées) par chacun des processeurs. Pour le diagnostic, fait sur le processeur #0, on émet une alarme qui donne simplement le nombre d'alarmes émises par processeur.
Ceci évite que de « louper » une alarme qui ne se serait produite que sur un processeur.

2 Détails d'implémentation

Remarque

Ce paragraphe est constitué de notes de développement et devrait permettre à un œil extérieur de comprendre comment cela a été fait.

2.1 État global de l'exécution

Il est nécessaire de stocker l'état des processeurs pour essayer d'arrêter proprement un maximum de calculs.

On doit stocker : ok/erreur, séparer proc #0/autres

Fonctions nécessaires :
- dire que tout est ok partout.
- dire qu'une erreur a été vue sur proc #0 ou autres.
- savoir si tout est ok.
- savoir si erreur sur proc #0 ou autres.

L'état est stocké dans un `COMMON` et deux routines existent pour interroger (`GTSTAT`, pour *get status*) et affecter l'état (`STSTAT`, pour *set status*). On utilise des constantes pour simplifier la lecture (voir `aster_constant.h`).

Contenu de `aster_constant.h` :

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

On utilise des opérations logiques bit à bit pour stocker et savoir si un état est positionné.

2.2 Communications non bloquantes

La clé pour détecter que certains processeurs ne répondent pas aux rendez-vous est d'utiliser des communications MPI non bloquantes.
Le point de départ du traitement spécifique est dans `u2mesg` lors de l'émission du message d'erreur.

2.2.1 u2mesg

En cas d'erreur, on prévient le proc #0 en appelant `mpicmw()`. Idem dans `Utmess.py`, en appelant, `aster_core.mpi_warn()`.

2.2.2 mpisst

Envoi au proc #0 `ST_OK` ou `ST_ER` (`MPI_ISEND` tag `CHK`, nonblocking send) et attend la réponse de proc #0 (`MPI_IRECV` tag `CNT`, nonblocking receive). On met un timeout pour ne pas attendre indéfiniment. Si le proc #0 ne répond pas dans le délai, on appelle `MPI_Abort` via `mpistp(1)`.

Si on a envoyé `ST_OK`, on veut savoir si on doit continuer ou pas.

Si on a envoyé `ST_ER`, on veut juste savoir si le proc #0 répond (dans ce cas on s'arrête proprement), sinon on doit interrompre l'exécution.

Si pas de timeout, on retourne la réponse de proc #0 : `ST_OK` (tout va bien), `ST_ER` (faire un arrêt propre).

2.2.3 mpicmw

Alerter le proc #0 qu'on a rencontré un problème

- Sur proc != 0, on positionne `ST_ER_OTH` (erreur sur un processeur autre que #0) et on envoie `ST_ER` au proc #0 avec `mpisst(ST_ER)`.

La réponse de proc #0 est `ST_ER`, on continue comme en séquentiel (exception, fermeture des bases ou abort).

- Sur proc #0, on positionne `ST_ER_PR0` (erreur spécifique au proc #0) et on appelle `mpichk()`.

2.2.4 mpichk

Appelé avant de faire une communication globale pour vérifier que tout va bien et sinon agir en conséquence.

- Sur proc != 0, on envoie `ST_OK` au proc #0 avec `mpisst(ST_OK)` et on attend la réponse du proc #0 pour savoir si on doit continuer ou interrompre.

  Si le proc #0 répond qu'il faut interrompre l'exécution, on appelle `mpistp(2)`.

- Sur proc #0, on attend la réponse de tous les autres processeurs (MPI_IRECV tag CHK nonblocking receive) + un timeout pour ne pas attendre indéfiniment.

  - Si un proc a rencontré une erreur (et donc envoyé ST_ER), message « erreur sur le proc #i » + STSTAT(ST_ER_OTH).

  - Si un des procs ne répond pas dans le délai, message « le proc #0 a trop attendu » et erreur 'E' « le processeur #i n'a pas répondu » + STSTAT(ST_UN_OTH).

  - Aux processeurs présents au rendez-vous, on répond 'continuer' ou 'interrompre' (MPI_SEND tag CNT, blocking send). En cas d'erreur sur le proc #0, on envoie interrompre. Pour interrompre, on appelle mpistp(2).

  - Si un des processeurs n'a pas été au rendez-vous, le proc #0 interrompt l'exécution avec MPI_Abort : on appelle mpistp(1).

mpichk fournit un code retour : 0 = ok, 1 = nook.

2.2.5 mpistp

Utilisé pour interrompre l'exécution.

- mpistp(2): tous les processeurs ont communiqué leur état, on peut donc interrompre proprement l'exécution avec u2mess('M', 'APPELMPI_95'). Si une exception a déjà été levée par u2mess ('F' ou 'S') précédent, il faut éviter la récursivité et ne pas lever une autre exception. Si aucune erreur n'a déjà été émise, le comportement est celui d'une erreur 'F' ordinaire.

- mpistp(1): au moins un processeur n'a pas répondu (peut-être le proc #0), on doit interrompre tout le monde y compris celui-ci qui ne répond pas. On émet un u2mess('D', 'APPELMPI_99') qui imprime le message avec 'F' (pour le diagnostic) mais n'émet pas d'exception -- ce qui provoquerait un débranchement, et on exécuterait donc pas la suite -- puis on appelle JEFINI('ERREUR') pour déclencher MPI_Abort.

- si ERREUR_F='ABORT', mpistp(2) devient mpistp(1).

- On ne devrait pas exécuter d'instruction après un appel à mpistp(2), faire GOTO fin de routine quand on appelle mpichk().

2.2.6 mpicm1 / mpicm2

Avant de commencer une communication, on appelle mpichk() pour vérifier qu'il n'y a pas eu de problème. Tenir compte du code retour et arrêter sans faire la communication !

2.2.7 jefini / MPI_Abort

Au lieu d'arrêter avec ABORT(), on appelle ASABRT(6) (6 correspond à SIGABRT) qui appelle MPI_Abort.

Il est indispensable d'appeler MPI_Abort pour être capable d'arrêter tout le monde, y compris les processeurs bloqués. Or MPI_Abort implique la fin du script lancé par mpirun et donc la copie des résultats du répertoire du proc #0 vers le répertoire global ne pourra pas avoir lieu (enfin, cela peut dépendre de l'implémentation MPI).
Donc « erreur en MPI » doit entraîner « pas de base sauvegardée » et en cas d'erreur, le diagnostic risque de ne pas être très détaillé (selon l'implémentation MPI, les fichiers fort.8/fort.9 sont ou ne sont pas recopiés dans le répertoire de travail global). Le diagnostic risque d'être <F>_ABNORMAL_ABORT au lieu de <F>_ERROR.

2.2.8 Notes supplémentaires, précautions

`MPI_Abort` n'interrompait pas l'exécution.

En MPI, il faut que les processeurs passent tous par `MPI_Finalize` avant de sortir.
Or dans l'interpréteur Python, on sort par '`sys.exit()`' qui appelle probablement la fonction système '`exit`' et donc on ne peut pas ajouter d'appel à `MPI_Finalize` avant de sortir. C'est pourquoi, on enregistre une fonction qui exécute `MPI_Finalize` via '`atexit`'.
Le problème est que cette fonction est aussi appelée après un `MPI_Abort`. L'exécution est donc bloquée sans interrompre tous les processeurs. On définit donc une fonction `ASABRT` qui fait '`abort`' ou '`MPI_Abort`' en parallèle et qui positionne un flag pour ne pas passer par `MPI_Finalize` dans la fonction '`terminate`' (cf. "`aster_error.c/h`").

Précaution pour les appels fortran depuis le C

Dès lors qu'on appelle des routines fortran depuis le C, sachant que quasiment toutes les routines sont susceptibles d'émettre des `u2mess` et donc de lever des exceptions, il est impératif que l'extension C du module (`aster` ou `aster_core`) prévoit un `try/except` (en C) pour traiter cette exception (et retourner `NULL` en cas d'erreur).
En effet, l'exception provoque un débranchement de l'exécution. S'il n'y a pas de `try`, on risque de ne pas être rebranché où l'on croit. Une erreur de programmation alerterait si aucun `try` n'a été mis en place plus haut.

Exemple :

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

2.3 Valeurs des délais

Il s'agit des délais accordés aux retardataires lors des communications non bloquantes.

Écart entre deux processeurs :

```
#0 ========|t0|......|ti|...

#i ==================|ti|...
```

$[ti-t0]$: délai accordé par #0 aux processeurs #i. Donc si #1 arrive avant #0, il doit lui accorder le même délai: $t0-t1=ti-t0$.

Le cas extrême est :

```
#0 ==================|t0|~~~~~~|ti|.v=========
                       ^          ^   v
#1 ========|t1|.......^~~~~~~~~~^~~~v=========
                                  ^
```

```
#i ============================|ti|~|tf|======
```

* $t1$: arrivée du premier processeur #1

* $t0$: arrivée de #0, #0 reçoit CHK de #1

* $t0+dt$: #1 attend la réponse de #0

* ti : arrivée de #i, #0 reçoit CHK de #i, #0 envoie CNT à #1 et #i

* $ti+dt=tf$: #1 et #i reçoivent CNT de #0

Il faut donc que $tf-t0>ti-t0$. On limite le délai de réception de la réponse de #0 à $1.2\times[ti-t0]$

La valeur du timeout est fixée à 20% du temps cpu restant.