Paralléliser en Esope dans Cast3M avec la bibliothèque pthread

Mise en place rapide en Esope

- Mettre l'include `-INC CCASSIS` qui contient les infos sur le nombre de threads `nbthrs`
- Déclarer la source d'interfaçage en `EXTERNAL` (ex : `EXTERNAL chole3i`)
- Déclarer un `COMMON` contenant un `SEGMENT` de travail que vont se partager les threads
- Ne pas paralléliser en pthread dans les ASSISTANTS : `CALL OOONTH(ITH)`
    - Appeler la `SUBROUTINE` qui fait le travail sans passer par le `COMMON` sinon il y aura des soucis
- Initialisation des threads et verrouillage : `CALL THREADII`
    - Le `COMMON` doit être un tableau pour accueillir les informations « par thread »
- Déterminer le nombre de threads utiles `NBTHR` (Boucles avec moins d'éléments que de threads)
- Boucle `ith` sur les threads de `2` à `NBTHR`
    - Lancement du travail en asynchrone : `CALL THREADID (ith, chole3i)`
- Lancement du thread maître séparément en dernier : `CALL chole3i(1)`
- Boucle `ith` sur les threads de `2` à `NBTHR`
    - Attente de la fin du travail : `CALL THREADIF(ith)`
- Stopper les threads : `CALL THREADIS`
    - **ATTENTION** : Ne pas faire de `RETURN` avant d'avoir fait les `THREADIF` et le `THREADIS`

Important

- Il n'est pas toujours possible de faire des `WRITE` de `REAL*8` dans les threads...
    - `WRITE` fonctionne dans le thread maitre
    - `PRINT` semble fonctionner dans les threads
- Il ne faut pas faire de **`SEGINI`**, **`SEGACT`**, **`SEGDES`**, **`SEGADJ`**, **`SEGSUP`** dans les threads
    - Créer / Activer tous les SEGMENTS avant l'appel et les laisser actifs durant le travail (On peut utiliser les SEGMENTS actifs sans soucis).
    - Désactiver / Supprimer les SEGMENTS après l'appel.
    - `OOV(1)` : Accessible par tous les threads. Indicateur de l'utilisation d'un `ELEMENT DE SEGMENT`
- Il ne faut pas créer de tableaux dynamiques
    - `REAL*8 XTOTO(NN)` ➔ `XTOTO` et `NN` doivent arriver en argument ou en `COMMON`. Un SEGMENT contenant `XTOTO` doit être préparé avant.
    - Avec les directives `IMPLICIT INTEGER(I-N)` et `IMPLICIT REAL*8(A-H,O-Z)` le type des SEGMENTS extensibles dépend aussi de la première lettre

Exemple de mise en application

```
      SUBROUTINE TOTO
      IMPLICIT INTEGER(I-N)
      IMPLICIT REAL*8 (A-H,O-Z)
C     Subroutine de préparation et lancement de TUTUi en parallèle
-INC CCASSIS
      EXTERNAL TUTUi
C     SEGMENT SPARAL : partagé par COMMON ou envoyé en argument
      SEGMENT SPARAL
        INTEGER NBTHR1
        INTEGER IPOINT1
        INTEGER IPOINT2
      ENDSEGMENT
      COMMON/TUTUC/IPARAL

      IPARAL = 0

      NBTHR= MIN(MAX(NT1/IOPTIM,1), NBTHRS)
      IF((NBTHR.EQ.1).OR.(NBTHRS.EQ.1).OR. (oothrd.GT.0)) THEN
        NBTHR = 1
        BTHRD =.FALSE.
      ELSE
        BTHRD =.TRUE.
C       Initialisation des threads
        CALL THREADII
      ENDIF

      IF(BTHRD) THEN
C       Préparation des donnes à partager dans SPARAL
        SEGINI,SPARAL
        SPARAL.NBTHR1= NBTHR //Donnees d'entrée pour TUTU1
        SPARAL.IPOINT1= ARG1
        SPARAL.IPOINT2= ARG2

C       Remplissage du COMMON/TUTUC
        IPARAL = SPARAL

C       Lancement du travail en parallèle
        DO ith=2,NBTHR
          CALL THREADID(ith, TUTUi)
        ENDDO
C       Lancement du travail sur le maitre
        CALL TUTUi(1)

C       Attente de la fin du travail des threads
        DO ith=2,NBTHR
          CALL THREADIF(ith)
        ENDDO

C       Stop des threads
        CALL THREADIS
C       Ménage des SEGMENTS de travail temporaires
        SEGSUP,SPARAL

      ELSE
C       Dans les ASSISTANTS ou en SEQUENTIEL on invoque directement la
C       SUBROUTINE qui fait le travail avec ses arguments
        ith=1
        CALL TUTU1(NBTHR,ith,ARG1,ARG2)
      ENDIF
      END
```

```
      SUBROUTINE TUTUi(ith)
      IMPLICIT INTEGER(I-N)
      IMPLICIT REAL*8 (A-H,O-Z)
C     Subroutine qui réalise l'interface en parallèle
      COMMON/TUTUC/IPARAL

C     SEGMENT SPARAL : partagé par COMMON ou envoyé en argument
      SEGMENT SPARAL
        INTEGER NBTHR1
        INTEGER IPOINT1
        INTEGER IPOINT2
      ENDSEGMENT

C     Récupération du SEGMENT dans le COMMON
      SPARAL = IPARAL
      NBTHR  = SPARAL.NBTHR1
      ARG1   = SPARAL.IPOINT1
      ARG2   = SPARAL.IPOINT2

      CALL TUTU1(NBTHR,ith,ARG1,ARG2)
      END
```

```
      SUBROUTINE TUTU1(NBTHL,ITHR,ARG1,ARG2)
      IMPLICIT INTEGER(I-N)
      IMPLICIT REAL*8 (A-H,O-Z)

      NNC = ARG1.BLABLA(/1) //Nombre de termes a découper

C     On assure le travail contigu en mémoire
      IF(NBTHL .EQ. 1)THEN
        IDEB  = 1
        IFIN  = NNC
      ELSE
        NBTHR=MIN(NNC,NBTHL)
        IF(ITHR .GT. NBTHR) ==> quitter / iterer la boucle
        IRES = MOD(NNC,NBTHR)
        IF(IRES  .EQ. 0)THEN
          ILON  = NNC / NBTHR
          IDEB  = (ITHR-1)* ILON + 1
        ELSE
          IF (ITHR .LE. IRES) THEN
            ILON  = (NNC / NBTHR) + 1
            IDEB  = (ITHR-1)* ILON + 1
          ELSE
            ILON  = NNC / NBTHR
            IDEB  = (IRES * (ILON+1)) + (ITHR-IRES-1)* ILON + 1
          ENDIF
        ENDIF
        IFIN  = IDEB + ILON - 1
      ENDIF

C     Boucle qui réalise le travail de IDEB à IFIN (Unrolling possible)
      DO J1 = IDEB,IFIN
        ARG2.BLIBLI(J1)= ...
      ENDDO
      END
```

Paralléliser dans Cast3M avec les ASSISTANTS

SUBROUTINES liées au travail dans les ASSISTANTS

- `chkesc.eso` : Vérifie si les opérandes contiennent des `TABLES ESCLAVES` et si une condensation de ces dernières est nécessaire (On y passe seulement si « `OPTI PARA VRAI ;`).
- `etg.eso` : OPÉRATEUR permettant de lancer la condensation d'une `TABLE ESCLAVE`
- `funobj.eso` : Effectue la condensation par tournoi d'une `TABLE ESCLAVE` avec un traitement particulier pour les `FLOTTANTS` et les `LOGIQUES`
- `assist.eso` : Prépare la panoplie complète des SEGMENTS (vierges) de `CCASSIS.INC` en attente de l'écriture des résultats par `nouins2.eso` (Travaille sur MAITRE)
- `nouins2.eso` : Écriture des résultats de chacun des ASSISTANTS dans le SEGMENT `NESRES` ➔ `MESRES` ➔ `esrees(iop)` où `iop` est le numéro de l'assistant (Travaille en parallèle)
- `acctab.eso` : Attend que le contenu de l'objet `ESCLAVE` soit disponible avant de le récupérer
- `oooprl.eso` : Place `oooprl(1)` ou retire `oooprl(0)` un `PERSISTANT_LOCK` pour un assistant. Cela permet de gérer des `SEGXXX` par paquets dans les ASSISTANTS sans générer des `GLOBAL_LOCK` qui entrainent de fortes attentes sur l'accès à GEMAT.

Faire un point de synchronisation

- En GIBIANE, afin d'attendre qu'un opérateur travaillant sur les ASSISTANTS ait terminé, on peut lancer un ménage obligatoire : `MENA 'OBLI' ;`

Remarque

- Le ménage peut supprimer des SEGMENTS de pré-conditionnement, il faut penser à les protéger (voir `menag6.eso`)
- La cohabitation des ASSISTANTS avec les PTHREAD n'est pas un problème pour des opérations unitaires qui utiliseraient les 2 méthodes en même temps (voir `threadid.c` dans `/u2/castem/castem.arc`). En pratique l'utilisation des 2 niveaux de parallélisme n'offrent pas de gain car les morceaux à traiter sont plus petits.
- Veiller à ce que les sources appelées dans les ASSISTANTS retirent bien le statut `*MOD` des SEGMENTS qu'elles ont créés sans quoi il faut s'attendre à des `DEADLOCK`. (Appeler `actobj` avant de faire `ecrobj`)
- Si une source a déjà activé les THREADS (`threadii`) et que cette opération est refaite plus loin, cela provoque un `DEADLOCK` non signalé par `GEMAT` (on est tout simplement bloqué…).
- Lorsqu'on veut faire un **`SEGADJ`** et qu'il faut récupérer la taille du SEGMENT en question avant, il faut prendre le SEGMENT en `*MOD` juste avant de récupérer la taille sans quoi celle-ci peut avoir été changée par un autre ASSISTANT entre temps !

Erreurs de programmation

Erreurs captées par le gestionnaire de mémoire ESOPE (GEMAT)

- Ce sont des erreurs captées par ESOPE lors de la manipulation de SEGMENTS.
- La SUBROUTINE en charge d'afficher le message d'erreur est `ooomes.eso` (Source ESOPE appelée par `oooerr.eso`)

`GEMAT ERROR : PAS ASSEZ DE PLACE EN MEMOIRE`

Ce message s'affiche lorsqu'il n'y a plus suffisamment de mémoire vive pour créer ou activer un SEGMENT.

- Le calcul est trop gros et l'erreur survient dans l'opérateur `'RESO'` en méthode directe. Il faut envisager de diminuer la taille du problème à résoudre ou de passer en résolution itérative (Voir `OPTI 'RESO' 'ITER';`)
- Il n'y a pas assez de place sur le disque pour le fichier de débordement ESOPE : Le système ne parvient plus à basculer sur le disque (SWAP) les SEGMENTS actuellement désactivés et ne dispose plus assez de mémoire vive pour gérer les nouveaux SEGMENTS.
- Un opérateur crée trop de SEGMENTS temporaires sans les supprimer ou les désactiver. La directive MENAGE ne peut pas les gérer car elle n'est invoquée qu'en sortie des opérateurs sous certaines conditions (voir `pilot.eso`). C'est probablement une anomalie : contacter le Support Cast3M.

`GEMAT ERROR : HORODATAGE INCORRECT`

Ce message s'affiche lorsqu'on essaye de supprimer un SEGMENT qui a été créé dans un opérateur précédent. Seule la directive MENAGE a les capacités de supprimer d'anciens SEGMENTS plus répertoriés.

- Évolution ESOPE : Depuis la version 2017 d'ESOPE, ce cas ne provoque plus d'erreur. Le SEGMENT n'est pas supprimé. Le MENAGE s'en chargera si nécessaire.

`GEMAT ERROR : LE POINTEUR DESIGNE UN SEGMENT SUPPRIME`

Cela signifie que le SEGMENT manipulé dans un SEGXXX fait référence à un descripteur ESOPE existant mais non actif. Le SEGMENT a été supprimé précédemment. Il faut surveiller le SEGMENT pour voir son historique.

Débusquer l'anomalie

- Il faut suivre le SEGMENT problématique afin de voir où il est supprimé
  - `OPTI 'SURV' NUMERO;`

`GEMAT ERROR : DESTRUCTION MEMOIRE`

Cela signifie que les valeurs inscrites dans les entêtes d'un SEGMENT sont incorrectes. Une SUBROUTINE a écrasé ces entêtes dans une boucle trop longue.

Débusquer l'anomalie

- La compilation des sources de Cast3M avec l'option « CONTROLE » d'ESOPE permet d'ajouter la vérification du non-dépassement des tailles des segments lors de chaque opération (C'est plus lent mais très efficace).
- Si l'écrasement survient dans une source en FORTRAN 77 pur, le travail de recherche est plus complexe.
- Il est possible d'invoquer `MENAGE` dans `pilot.eso` de manière systématique après chaque opérateur. Il suffit pour cela de mettre `IFMEN=5` juste avant la ligne « `IF(IFMEN.NE.0)THEN` ». Il faut faire la même modification dans `menage.eso`. `MENAGE` fait un test sur tous les SEGMENTS et si l'un d'entre eux est invalide cela sera affiché par un message : `ajoun incorrect` (extrêmement lent).

GEMAT ERROR : ARGUMENT(S) ELEMENT(S) DE SEGMENT

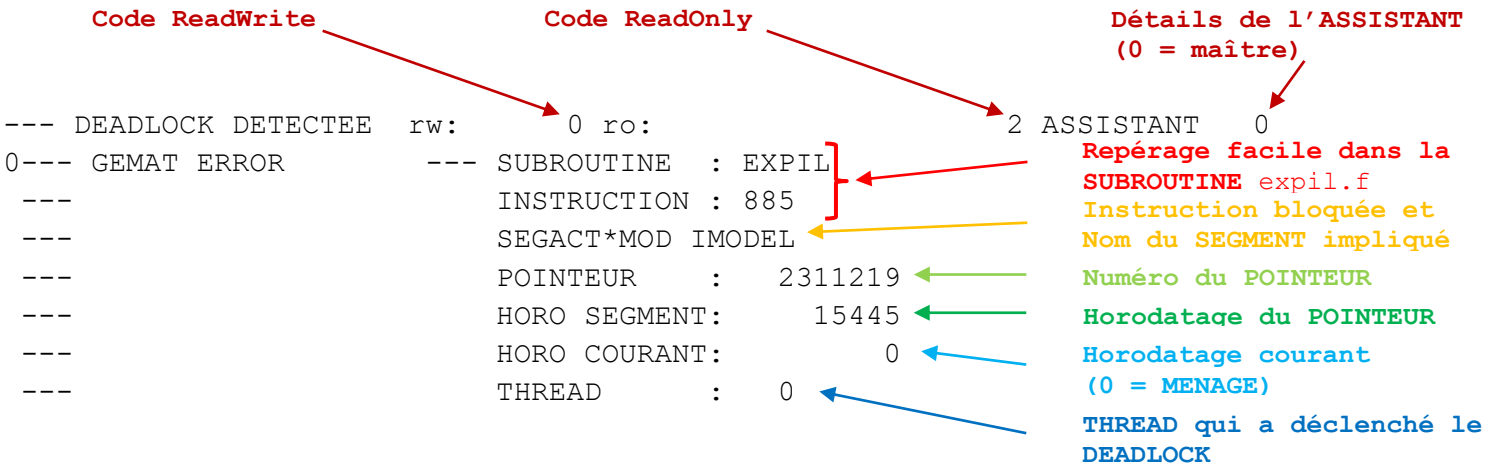
Cela signifie que dans l'arbre d'appel conduisant à l'erreur, un élément de segment est envoyé comme un argument d'une SUBROUTINE et que cette dernière utilise des directives ESOPE (SEGINI, SEGACT, SEGDES, SEGSUP). Il faut rechercher la SUBROUTINE traduite qui possèdes dans ses arguments des OOT, OOA, OOO(…). Il faut envisager de passer le SEGMENT et non pas ses arguments.

GEMAT ERROR : DEADLOCK DETECTE

La SUBROUTINE en charge de détecter le DEADLOCK est oooddl.eso (Source ESOPE)

Comprendre le message affiché

En travaillant avec les ASSISTANTS, il est courant de faire face à des DEADLOCK. Cette situation survient lorsque tous les ASSISTANTS sont en attente d'un SEGMENT ESOPE. Tant qu'au moins un ASSISTANT n'est pas bloqué, on n'est pas considéré en DEADLOCK. L'exemple ci-dessous permet de comprendre comment lire le message d'erreur que renvoie Cast3M :



Débusquer l'anomalie

Afin de trouver la source qui a pris le SEGMENT IMODEL en lui laissant le statut « écriture » (*MOD), voici la procédure à suivre :

- **Cas simple** : Surveiller l'activité sur un SEGMENT puis relancer le cas-test
  - 'OPTI' 'SURV' 2311219 ; pour placer une surveillance dans le jeu de données GIBIANE
  - CALL OOOSUR(2311219) pour placer une surveillance dans une source ESOPE
  - Chaque directive ESOPE (SEGINI, SEGACT, SEGDES, SEGSUP) qui manipule le SEGMENT surveillé affiche les informations présentées ci-dessus. Il suffit de trouver en partant de la fin quelle SUBROUTINE a créée le SEGMENT, pour ensuite remonter à l'opérateur qui ne l'a pas désactivé avant de rendre l'objet sur la pile (il manque un ACTOBJ quelque part).

- **Cas un peu plus compliqué** : Les numéros de pointeurs changent à chaque exécution de Cast3M lors du DEADLOCK. C'est dû au fait qu'en parallèle les opérations ne se font pas nécessairement dans le même ordre.
  - Récupérer pilot.eso dans l'archive
  - Retirer les commentaires des lignes de test avant l'aiguillage pour les opérateurs (~lignes 427)
  - Dans le test, mettre l'entier de l'horodatage du SEGMENT indiqué dans le DEADLOCK (ici : 15445)
    - Eventuellement ajouter un CALL TRBAC (pour afficher l'instruction en question)
  - Compiler pilot.eso et faire l'édition des liens
  - Lors de l'exécution du cas-test, l'opérateur qui va écrire en sortie l'objet « vérolé » sera imprimé à l'écran (avec quelques autres informations)

- **Cas plus compliqué encore** : La méthode précédente échoue car l'objet est créé hors opérateur
    - C'est le cas de `chkesc.eso` qui peut fusionner des TABLES et rendre des objets sur la pile
    - Invocation d'une procédure ou d'une méthode (ne place pas d'objets dans la pile)
    - Ménage automatique (ne place pas d'objets dans la pile)
    - Toujours dans `pilot.eso`,
        - Mettre le test sur l'horodatage juste après sa définition (~ligne 229).
        - Eventuellement ajouter des `CALL TRBAC` pour les deux ou trois instructions qui précèdent sur cet ASSISTANT (aide à retrouver la commande GIBIANE en question)
            - `IF(IHORO .EQ. iVal - (NBTHRS + 1) * 1) CALL TRBAC`
            - `IF(IHORO .EQ. iVal - (NBTHRS + 1) * 2) CALL TRBAC`
            - `iVal` est l'entier correspondant à l'horodatage de l'objet « vérolé »

Erreurs conduisant à l'émission de signaux du système

- `SIGABRT` : Signal renvoyé par exemple par `malloc()` ou `free()` lors de troubles sérieux (comme un overflow dans la mémoire demandée). La fonction invoquée est `abord()` (`abort` sous windows).
- `SIGSEV` : Signal indiquant une tentative d'accès invalide à une adresse mémoire valide (ex : essayer d'écrire là où on ne peut que lire).
- `SIGBUS` : Signal indiquant une tentative d'accès à une adresse mémoire invalide (overflow). C'est un signal différent de SIGSEV.
- `SIGFPE` : Signal indiquant une erreur arithmétique fatale (division par zéro, overflow, NaN, etc.)
- `SIGILL` : Signal indiquant la tentative d'exécution d'une instruction illégale. Le CPU tente d'exécuter une instruction qu'il ne connait pas. Saut à une adresse qui n'est pas dans l'aire définie par le programme. Compilation pour une mauvaise architecture. Ne rien retourner alors qu'on a spécifié un retour, ect.

Erreurs conduisant à un blocage sans DEADLOCK

Il arrive parfois d'être bloqué sans que la main ne soit rendue par Cast3M et pour autant il n'y a pas de `DEADLOCK DETECTE`. Dans ce cas, il suffit de récupérer la main sur le processus bloqué (dans un autre terminal) avec `gdb` et de demander aux `threads` où ils sont en ce moment :

```
ps -edf | grep cast
USERxyz 25759 25725 0 14:48 pts/5 00:00:00 ./cast
/usr/bin/gdb -p 25759
thread apply all where
```

Commandes gdb utiles

- Exécuter Cast3M avec `gdb (castem -d cas-test.dgibi)`
- Commande `run`
  - Exécute Cast3M dans l'environnement `gdb`
- Commande `break | symbol_ | Num_ligne | Instruction` (`stop in` sur IBMRS6000)
  - Arrêt du programme à chaque entrée dans la subroutine *`symbol_`*
  - Arrêt du programme à la ligne FORTRAN *`Num_ligne`* de la SUBROUTINE courante
  - Arrêt du programme lors de l'exécution de l'instruction FORTRAN demandée (ex : `ITOTO=3`)
- Commande `where` (ou `thread apply all where` en parallèle)
  - Affiche l'arbre d'appel au moment de l'arrêt du programme (`backtrace`)
- Commandes `up` et `down`
  - Remonte ou descend dans l'arbre d'appel
- Commandes `l` *`(Num_ligne)`*
  - Sans *`Num_ligne`*, Liste le FORTRAN autour de la ligne où `gdb` s'est arrêté
  - Avec *`Num_ligne`*, liste le FORTRAN autour de la ligne demandée
- Commandes `c`
  - Continuer l'exécution du programme après une interruption

SUBROUTINES utiles pour travailler sur les OBJETS en Esope

SUBROUTINES ESOPE

`oooeta.eso` ➔ Donne l'état d'un `SEGMENT` (`PSEG`, `IETA`, `IMOD`) pour le thread courant (`oothrd`)
`IETA=1` (Actif) | `IMOD=0` (Lecture seule)
`IETA=2` (Inactif) | `IMOD=1` (Écriture)

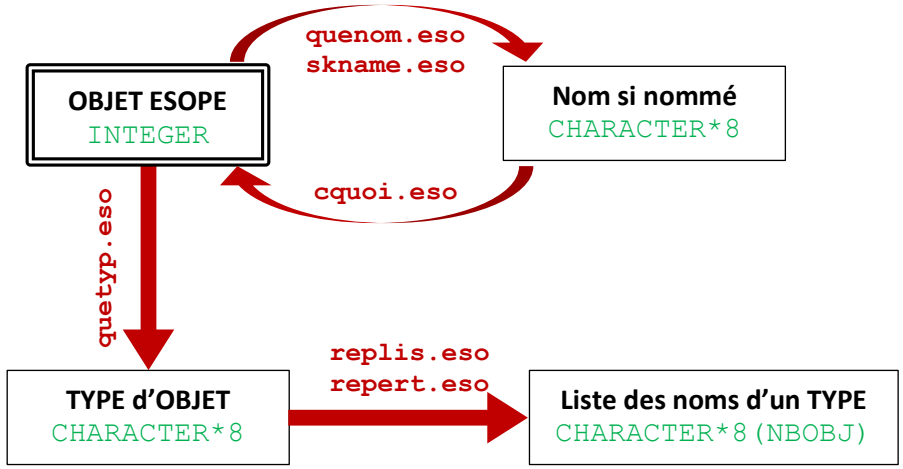
`ooove1.eso` ➔ Donne la validité d'un `SEGMENT(LRET,POINT)`
`LRET=1` (Invalide)
`LRET=2` (OK)
`LRET=3` (Supprimé)

`ooohor.eso` ➔ Défini l'horodatage courant pour un thread donné
`oooho1` → `ENTRY(ISEG,IHOR)` dans `ooohor` pour récupérer l'horodatage d'un `SEGMENT`
`oooho2` → `ENTRY(IHOR)` dans `ooohor` pour récupérer l'horodatage courant pour un thread donné

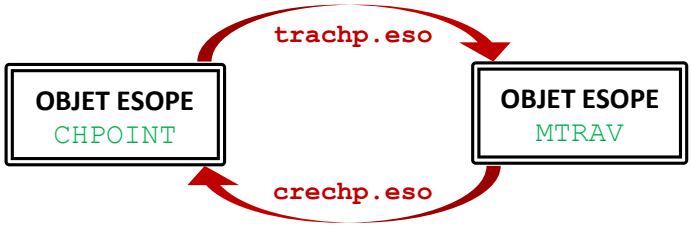
`ooomut.c` ➔SUBROUTINE qui traite les MUTEX (entre autres choses)
`oooprl` → Traite les PERSISTANT LOCK (`IGLL`)
`IGLL=1` (prend le LOCK)
`IGLL=0` (libère le LOCK)
`oooprs` → Donne l'info si le PERSISTANT LOCK est déjà pris (`IPRS`)
`IPRS =1` (le LOCK est pris)
`IPRS =0` (le LOCK est libre)

SUBROUTINES Cast3M

intpdo.eso ➔ Donne combien il faut d'INTEGER pour faire l'équivalence avec un REAL*8 (32/64-bits)
actobj.eso ➔ Active/Désactive tous les SEGMENTS contenus dans un objet donné en argument
quenom.eso ➔ Donne le NOM (CHARACTER*8) du dernier objet lu (voir lirobj.eso)
skname.eso ➔ Donne le 1er nom trouvé (CHARACTER*8) pour un pointeur (INTEGER)
cquoi.eso ➔ Donne le POINTEUR (INTEGER) d'un objet dont on connait le NOM (CHARACTER*8)
quetyp.eso ➔ Donne le TYPE (CHARACTER*8) du premier objet dans la PILE sans le retirer
replis.eso ➔ Répertorie tous les NOMS d'un TYPE d'objet dans un SEGEMENT de CHARACTER*8
repert.eso ➔ Ecris dans la pile GIBIANE tous les OBJETS du type demandé
quidan.eso ➔ Ne conserve que les MAILLAGES d'une liste dont les nœuds sont inferieurs a IMAX
melinc.eso ➔ Répertorie tous les MAILLAGES nommés inclus dans le MAILLAGE donné en argument



rmpgbn.eso ➔ Remplace dans la pile GIBIANE un objet par un autre. Remplace également selon la valeur d'un LOGIQUE l'objet de manière permanente (pour ne pas avoir à refaire systématiquement la fusion des TABLES ESCLAVES)
poscha.eso ➔ Position d'un MOT dans la pile des MOTS
poslog.eso ➔ Position d'un LOGIQUE dans la pile des LOGIQUE
posree.eso ➔ Position d'un FLOTTANT dans la pile des FLOTTANT
trbac.eso ➔ Permet de renvoyer l'ECHO de la ligne GIBIANE en cours de traitement (Plante si la ligne contient des objets temporaires…) ;
typfil.eso ➔ Établit la correspondance entre un TYPE d'objet et son numéro de PILE
savseg.eso ➔ Permet de protéger des SEGMENTS que l'on ne souhaite pas voir supprimés par MENAGE **Attention**, il faut s'assurer qu'après un SAUV/REST cela fonctionne toujours !
libseg.eso ➔ Permet de retirer des SEGMENTS de la liste des SEGMENTS protégés (par savseg.eso)
GIBI.ERREUR➔ Contient tous les messages d'erreurs dans toutes les langues (FRAN et ANGL actuellement). Pour l'évoluer sur semt2 avec dial20 il faut le renommer gibi.erreur en minuscule (C'est historique).
trachp.eso ➔ Convertit un CHPOINT en SEGMENT de travail MTRAV
crechp.eso ➔ Convertit un SEGMENT de travail MTRAV en CHPOINT. Il faut compléter le chapeau après.



SEGXXX d'un SEGMENT quelconque

Pour manipuler un SEGMENT ESOPE avec GEMAT sans s'occuper de son contenu (voir `menag4.eso`) :

- Définir un SEGMENT de travail :
  - Travailler avec des **INTEGER** `SEGMENT ISEG(0)`
  - Travailler avec des **REAL*8** `SEGMENT XSEG(0)`
- Affecter ce SEGMENT d'un POINTEUR : `ISEG=IPOIN`
- Manipuler ce SEGMENT : `SEGSUP,ISEG` ou `SEGACT,ISEG` ou `SEGDES,ISEG`

Savoir en ESOPE le nom de la procédure courante

Le travail est fait dans la `SUBROUTINE finpro.eso` (voir détail ci-après).

```
C Includes ESOPE nécessaires
-INC SMBLOC
-INC CCNOYAU

C déclarations
      CHARACTER*(LONOM) NOMPRC

C travail de recherche de la procédure dans la liste des OBJETS
      DO 10 I=1,LMNNOM
         IF (INOOB1(I).EQ.1)               GOTO 10
         IF (INOOB2(I).NE.'PROCEDUR')      GOTO 10
         IF (IPIPR1(IOUEP2(I)).NE.MBLPRO) GOTO 10
         IP=INOOB1(I)
         IDEBCH=IPCHAR(IP)
         IFINCH=IPCHAR(IP+1)-1
         NOMPRC=ICHARA(IDEBCH:IFINCH)
         GOTO 11
   10   CONTINUE
   11   CONTINUE
```

Créer un nouvel opérateur dans Cast3M

Modifier la SUBROUTINE `pilot.eso`

1ère couche : SUBROUTINE sans ARGUMENTS ➔ Décodage des OBJETS GIBIANE

Râteau pour les différentes options (un seul `CALL LIRMOT` de préférence)

Lecture des OBJETS GIBIANES en ESOPE

`LIROBJ`
`REDUAF` (si lecture d'un `MMODEL` et `MCHAML` sauf pour l'opérateur `PROI`)
`ACTOBJ` (permet d'activer/désactiver le contenu d'un OBJET)
`LIRREE`
`LIRENT`
`LIRLOG`
`LIRMOT`
`ACCTAB`

Écriture des OBJETS GIBIANES en ESOPE

`ECROBJ`
`ECRREE`
`ECRENT`
`ECRLOG`
`ECRCHA`
`ECCTAB`
`ACTOBJ` (permet d'activer/désactiver le contenu complet de certains OBJETS Cast3M)

2ème couche : SUBROUTINES avec ARGUMENTS

Pour être appelé directement en `ESOPE` depuis une autre `SUBROUTINE`
Ne jamais passer les arguments par `COMMON` à moins qu'ils soient passés par thread.
Fait le travail, éventuellement en parallèle
Revoie un code de retour (`IRET`)

```
IRET = 1 si l'opération a été réalisée correctement
IRET = 0 si l'opération a rencontré une erreur
```

Teste en sortie le niveau d'erreur (si `Ctrl + C` a été pressé par exemple) attention à pas le tester dans les boucles les plus profondes (lent si tous les threads veulent y accéder en même temps)

```
IF (IERR .NE. 0) RETURN
```

Travailler à `SEGMENTS` ouverts de partout et ne désactiver que les `SEGMENTS` nouveaux (ou retirer le statut `*MOD`). Cela évite du `SPIN_LOCK` dans `ESOPE`.

Remarque : Travailler avec des arguments d'entrée qui sont des objets `TABLE` casse le parallélisme utilisant les ASSISTANTS.

- La `TABLE` avec `IPILOC` et `MCOORD` sont les seuls SEGMENTS dont le contenu est modifié sans changement du numéro de pointeur.
- Afin que les ASSISTANTS n'accèdent pas en même temps dans cette dernière, un « `GLOBAL_LOCK` » est pris lorsqu'on y accède via la `SUBROUTINE ACCTAB`.
- Si le contenu de l'objet `TABLE` doit être accédé en lecture seule, il convient d'accéder soi-même aux indices qu'elle contient.
- Les `SEGMENTS IPILOC` et `MCOORD` doivent être désactivé dès que possible sans quoi des « `GLOBAL_LOCK` » seront pris en attendant que ces SEGMENTS soient à nouveau libre…

Rédaction d'une NOTICE pour un opérateur / procédure

- 1ère ligne : `$$$$ LE_NOM  NOTICE` (Compter 10 caractères entre le dernier `$` et le `N` de `NOTICE`)
- Dernière ligne : `$$$$`
- Ligne séparatrice indiquant que le texte qui suit est en Français / Anglais :
  ```
  FRAN=======================================================
  ANGL=======================================================
  ```
- Découpage en CHAPITRES et PARTIES des notices pour un affichage plus clair avec INFO (Voir mode.notice) :
  ```
  CHAP{Nom de mon premier CHAPITRE}
  PART{Nom de la 1ère sous-partie de ce CHAPITRE}
  ```
- Si des Opérateurs / Procédures sont en relation, bien penser à les mettre dans la rubrique `Voir aussi`

Séminaire SEMT : Bonnes pratiques en FORTRAN

- **ATTENTION** : `READ` dans un fichier `ASCII` le comportement diffère suivant la plateforme :
  - IBM Aix : Les caractères `CR` et `LF` sont laissés comme tel, `READ` peut échouer
  - Linux/Windows: Les caractères `CR` et `LF` sont remplacés par un espace `' '`
- Comparaison de `CHARACTER` : Le compilateur `GFORTRAN` appelle `_gfortran_compare_string` dès qu'il s'agit de comparer des chaines de caractères de longueurs différentes. C'est extrêmement lent…
  - Recopier les chaines dans des chaines fixes déclarées en début de `SUBROUTINE`
  - Ajouter des espaces afin d'atteindre la bonne longueur (`MOT4.EQ.'MOT'` ➔ `MOT4.EQ.'MOT '`)
- Division entière `2x` plus rapide que la division par un flottant
- Pour diviser un tableau par une constante `C1`, il faut faire au préalable `C2=1.D0/C1` dans une variable et multiplier les valeurs du tableau par `C2`
- Puissances flottantes `16x` plus lentes que les puissances entières
  - Remplacer `x**(3.D0/2.D0)` par `SQRT(x**3)` est `4,5x` plus rapide
- Privilégier la continuité de la mémoire dans la lecture des tableaux (`2,5x` plus rapide dans le bon sens)
  - `Tab1(I,J,K)` est continue en mémoire d'abord sur `I`, puis sur `J` et enfin sur `K`
    - La boucle extérieure sur `K` et intérieure sur `I`
- Options de compilation recommandées en mode RELEASE :
  - `-funroll-loops`
  - `-funroll-all-loops`
- Options de compilation recommandées en mode DEBUG :
  - `-fbacktrace` : Permet de remonter l'arbre d'appel
  - `-Wunused-parameter` : Pour la lisibilité des sources
  - `-Wunderflow`
  - `-fbounds-check` : Vérifie les bornes des tableaux FORTRAN. Ne fonctionne pas avec ESOPE (à cause des `OOA(… OOO(…,2**30))`
  - `-O0 -g`
  - `-ftrapv` : Le mode de capture des signaux est verbeux
  - `-ffpe-trap=invalid,zero,overflow,denormal`
  - `-fimplicit-none` : Incompatible avec le traducteur ESOPE qui n'écrit pas les types des variables dont il se sert. Ceux-ci sont implicites !
- Utilisation de VALGRIND :
  - `Memcheck` : Vérification de la mémoire RAM : `--tool=memchek`
  - `Cachegrind` : Vérification de la mémoire CACHE : `--tool=cachegrind`
  - `Massif` : Profiler d'allocation mémoire : `--tool=massif`
  - `Helgrind` : Détecteur d'erreur dans les THREAD : `--tool=helgrind`
  - `DRD` : Détecteur d'erreur dans les THREAD : `--tool=drd`
  - `Iogrind` : Vérifications des Entrées / Sorties
  - `Wine` : Considéré comme illégal par Microsoft (Reverse Engineering sur les librairies)

- VALGRIND avec Cast3M :
  - Mettre `ZERMEM=VRAI` dans la variable d'environnement `ESOPE_PARAM`
  - Utiliser `ooozmr.eso` et non pas `ooozmr.c` (Sinon la mémoire est déclarée non initialisée…)
  - Exécuter VALGRIND sur `./cast`

Analyse des performances du code : PERF

Lancement d'analyses

- Temps CPU en `direct` : `perf top` (donne des infos par `SUBROUTINE` en « runtime »)
- Pour avoir l'assembleur annoté des lignes FORTRAN, utiliser `compils` sur `/u2`
- Enregistrement :
  - Rapide (sans options)
    ```
    perf record castem Test.dgibi
    ```
  - Avec l'arbre d'appel pour les instructions *User* seulement :
    ```
    perf record --call-graph lbr -e instructions:u castem Test.dgibi
    ```
- Post-traitement :
  - `perf report --call-graph=fractal,[caller|callee]` (appelant / appelé)
  - `perf report --no-children` (tri par durée propre `self`)
- Comparer 2 appels à `perf report` successifs :
  - `perf diff --sort=symbol`

Post traitement graphique

- Possibilité d'utiliser l'outil gratuit `hotspot` pour visualiser facilement les résultats

Pour Repérage des prises de lock par ESOPE

- Augmenter le spinlock (dans `ooomut.c`) : Variable d'environnement dans ESOPE pour transformer le temps d'attente du lock en temps CPU (Utile seulement avec les ASSISTANTS, en séquentiel il n'y pas d'effet).
  - `export ESOPE_SPINLOCK=ENTIER1.`
  - `ENTIER1` correspond à la longueur de la boucle de `SPINLOCK` : effets visibles à partir de 10000
  - Remplace les appels à `pthread_mutex_lock` par `pthread_mutex_trylock` qui consomme du temps CPU comptabilisable par « `perf` ».

Repérer dans une source possédant de nombreux `SEGXXX` lequel prend du temps

- Dans les sources ESOPE correspondantes (`ooowin.eso`, `ooowac.eso`, `ooowde.eso` et `ooowad.eso`) il faut positionner des `ENTRY` qui permettrons de distinguer les appels (certaines en ont déjà).
- Dans le `.f` de la source qui pose problème, il faut remplacer les `CALLOOOWxx` par les `ENTRY` dans ESOPE
  - Pour `SEGINI` : `CALLOOOWIN`
  - Pour `SEGACT` : `CALLOOOWAC`
  - Pour `SEGDES` : `CALLOOOWDE`
  - Pour `SEGADJ` : `CALLOOOWAD`
- Lancer l'analyse avec `perf` (Le temps CPU par `ENTRY` sera calculé)

Analyse de couverture du code : GCOV (avec GCC)

Commandes

- Compilation : `compilcast --gcov *.eso`
- Édition des liens : `essaicast`
- Exécution des cas-tests : `castem *.dgibi`
- Post-traitement : `gcov -b *.f > BILAN.txt`
- Rapport HTML : `lcov.bat` (il faut également avoir `perl` d'installé) mais marche PO…