# Mesurer la mémoire utilisée par *Code_Aster*

**Résumé :**
On présente quelques outils permettant de mesurer la mémoire utilisée dans les routines fortran ou les sources Python.

# Table des matières

# 1 Paramètres importants

## 1.1 Paramètres d'ASTK

```
Case "Mémoire Totale (Mo)" : MTOT
Case "Dont Aster     (Mo)" : MJEV
```

**MJEV**

MJEV est la mémoire qui sera fournie à JEVEUX pour l'allocation des objets "dynamiques" (un objet dynamique est un objet dont la taille est supérieure à la limite fixée dans DEBUT/MEMOIRE/DYNAMIQUE=xxx).

**MTOT**

MTOT est la mémoire totale qui sera allouée au processus. En interactif, ce paramètre est sans effet. En revanche, sur le serveur aster, le gestionnaire de ressources (RMS) contrôle que le processus ne dépasse pas cette limite.

MTOT doit être > MJEV

***Remarque concernant RMS***

*La mémoire utilisée par un processus est "multiple" : données de l'utilisateur, buffers d'entrée-sortie, exécutable et bibliothèques dynamiques, …*
*L'expérience a montré que pour lancer Code_Aster sur le serveur aster (version NEW9 du début 2008), la "prise en charge" du job : chargement de l'exécutable, mise en place de l'environnement d'exécution, ... consomme plusieurs centaines de Mo de mémoire avant même de pouvoir commencer à traiter les données de l'utilisateur.*
*C'est pourquoi, le gestionnaire de batch (LSF) demande à RMS d'autoriser le processus à consommer :*

*MEM_RMS = MTOT + 300 Mo*

**Signification de MTOT**

MTOT est donc la quantité de mémoire allouée au processus pour les besoins de l'utilisateur :

- Un calcul "2 fois plus gros" nécessitera en principe un MTOT double.
- Un tout petit calcul doit pouvoir passer avec MTOT ~ 0

**A quoi sert MTOT ?**

On a vu que la mémoire JEVEUX MJEV était prise sur MTOT. Le reste (MTOT - MJEV) est disponible pour :

- L'espace des variables python du jeu de commande
- Les allocations dynamiques faites dans le code : utilisation de Mumps, PETSc, ...

## 1.2 Paramètres de la ligne de commande

Sur la ligne de commande d'Aster, on peut trouver 2 paramètres concernant la gestion de la mémoire:

- -mem_jeveux mjevs (Mw)
- -mxmemdy mjevd (Mw)

Ces 2 paramètres sont exprimés en Mega-"mot" (Mw). Le "mot" étant l'espace mémoire nécessaire au stockage d'un INTEGER fortran (8 octets sur les machines 64 bits comme le serveur aster, 4 octets sur les machines 32 bits).

-memjeveux mjevs

mjevs est obligatoire. Il sert à dimensionner la mémoire "dynamique" JEVEUX. La valeur transmise par ASTK est celle déduite de MJEV (en changeant juste d'unité).

-mxmemdy mjevd

mjevd est facultatif. Il sert à limiter la mémoire "dynamique" JEVEUX. L'allocation dynamique JEVEUX s'arrêtera en erreur <S> dès que la longueur cumulée des objets dynamiques présents en mémoire dépassera cette limite. Pour l'instant, cette valeur ne peut être transmise à Code_Aster qu'en ajoutant le paramètre -mxmemdy sur la ligne de commande. On peut le faire avec ASTK dans le panneau "ETUDE" au bas de la fenêtre dans la zone de saisie "Arguments".
Par exemple, pour que la mémoire dynamique de JEVEUX soit limitée à 100 Mo, sur une machine "I8" comme la Bull, il faut écrire:

```
-mxmemdy 12.5
```

# 2 Que peut-on mesurer ? avec quels outils ?

## 2.1 Chiffres donnés par Aster

Dès le début d'un calcul, JEVEUX imprime dans le fichier "message" plusieurs informations concernant la gestion de la mémoire :

```
 MEMOIRE IMPOSEE POUR JEVEUX:         3145728 OCTETS (     3.000 MEGAOCTETS)
 MEMOIRE DONNEE PAR "MEMDIS":         3145728 OCTETS
 MEMOIRE PRISE               :        3145728 OCTETS (     3.000 MEGAOCTETS)
 LIMITE MEMOIRE DYNAMIQUE    :     1258291200 OCTETS (  1200.000 MEGAOCTETS)
```

Dans cet exemple, les informations importantes sont à la première et la dernière ligne : la mémoire statique JEVEUX est limitée à 3 Mo et la mémoire dynamique est limitée à 1200 Mo. Par défaut, on alloue tous les objets en mémoire dynamique et la mémoire statique est limitée à 1 Mo. Ainsi si on demande 100 Mo de mémoire totale dans ASTK, on aura 1 Mo sur la première ligne et 99 Mo sur la dernière.

A la fin du calcul, JEVEUX imprime d'autres informations concernant l'usage de la mémoire dynamique:

```
STATISTIQUES CONCERNANT L'ALLOCATION DYNAMIQUE :
     TAILLE CUMULEE MAXIMUM                    577  Mo,
     DONT                       3  Mo POUR LA ZONE GEREE PAR JEVEUX.
     TAILLE CUMULEE LIBEREE                   1173  Mo.
     NOMBRE TOTAL D'ALLOCATIONS  :                 487 .
     NOMBRE TOTAL DE LIBERATIONS :                 487 .
                          0  APPELS AU MECANISME DE LIBERATION.
     TAILLE MEMOIRE CUMULEE RECUPEREE :                  0  Mo.
```

Dans l'exemple précédent, on peut s'apercevoir (a posteriori) que la mémoire dynamique attribuée à JEVEUX (1200 Mo) est sur-dimensionnée : Avec 577 Mo, le calcul serait allé au bout sans déclencher le "mécanisme de libération". Il est probable que ce calcul puisse être fait avec moins de mémoire que 577 Mo.

## 2.2 La commande Unix "top"

La commande Unix top donne des informations (rafraîchies à intervalle régulier) sur le déroulement d'un processus.

Les 3 colonnes VIRT, RES, SHR concernent l'usage de la mémoire.

VIRT : total de la mémoire virtuelle utilisée par le processus.
RES : mémoire "résidente" (non-swappée)
SHR : mémoire "partagée"

Mais d'autres paramètres sont accessibles : DATA, SWAP, ...

Pour plus de détails, consulter man top

### 2.2.1 Le système de fichier /proc

Le système UNIX crée pour chaque processus de numéro xxxxx, un répertoire (temporaire) de nom /proc/xxxxx.

Dans ce répertoire, on trouve en particulier le fichier status. Ce fichier, remis à jour périodiquement par le système donne des informations sur l'usage de la mémoire par le processus. Les items intéressants ont un nom commençant par Vm (Virtual Memory).

On trouvera par exemple :

```
VmPeak:   174912 kB
VmSize:   168384 kB
VmLck:         0 kB
VmHWM:     33728 kB
VmRSS:     33664 kB
VmData:   159424 kB
VmStk:      1344 kB
VmExe:       384 kB
VmLib:      3520 kB
VmPTE:       704 kB
```

Une petite expérience sur la machine Bull a montré que les 3 paramètres :

```
VmSize
VmRSS
VmData
```

s'incrémentaient (et se désincrémentaient) après les ALLOCATE / DEALLOCATE de fortran 90.
Ils peuvent donc servir à "suivre" dans le temps la somme des variables allouées dynamiquement par le programme.

Pour plus d'informations : http://okki666.free.fr/docmaster/articles/linux070.htm

## 2.3 Mesurer l'usage de la mémoire dans un programme Fortan

Depuis la version 9.3.11, on connaît après chaque commande la consommation instantanée de JEVEUX, mais aussi le pic atteint jusque là :

```
#  USAGE DE LA MEMOIRE JEVEUX
#  - MEMOIRE DYNAMIQUE CONSOMMEE : 112.76 Mo (MAXIMUM ATTEINT : 558.45 Mo)
```

Par ailleurs, une indication intéressante à regarder en fin d'exécution est la mémoire minimale requise pour faire tourner le calcul (valeur MAXIMUM ATTEINT de la MEMOIRE UTILISEE) :

```
#     - MEMOIRE UTILISEE :    7.54 Mo (MAXIMUM ATTEINT :   110.47 Mo)
```

## 2.4 Mesurer l'usage de la mémoire dans un programme Python

Sur le Web, on a trouvé un petit morceau de code Python (voir ci-dessous) qui exploite le fichier /proc/xxxxx/status présenté au paragraphe précédent.

Dans un programme Python, si l'on souhaite connaître la consommation de mémoire d'un morceau de code, on peut faire :

```
mav=memory()
...
bout de code à instrumenter
...
map=memory()
print "Accroissment de la mémoire utilisée (en octets) :", map - mav
```

```
#-------------------------------------------------------------------------------
# Script Python pour mesurer l'utilisation de la mémoire :
#-------------------------------------------------------------------------------
import os

_proc_status = '/proc/%d/status' % os.getpid()

_scale = {'kB': 1024.0, 'mB': 1024.0*1024.0,
          'KB': 1024.0, 'MB': 1024.0*1024.0}

def _VmB(VmKey):
    '''Private.
    '''
    global _proc_status, _scale
     # get pseudo file  /proc/<pid>/status
    try:
        t = open(_proc_status)
        v = t.read()
        t.close()
    except:
        return 0.0  # non-Linux?
     # get VmKey line e.g. 'VmRSS:  9999  kB\n ...'
    i = v.index(VmKey)
    v = v[i:].split(None, 3)  # whitespace
    if len(v) < 3:
        return 0.0  # invalid format?
```

```
     # convert Vm value to bytes
    return float(v[1]) * _scale[v[2]]


def memory(since=0.0):
    '''Return memory usage in bytes.
    '''
    return _VmB('VmSize:') - since


def resident(since=0.0):
    '''Return resident memory usage in bytes.
    '''
    return _VmB('VmRSS:') - since


def stacksize(since=0.0):
    '''Return stack size in bytes.
    '''
    return _VmB('VmStk:') - since
#--------------------------------------------------------------------------------
```