

# Débogage HPC

**Intervenant·e·s** : Olga Abramkina, Isabelle Dupays, Thibaut Véry

22/11/2024



Image : copyright Photothèque CNRS/Cyril Frésillon

- 1 Introduction au débogage parallèle**
- 2 Quelques principes de débogage parallèle**
  - Modèle mémoire partagée
  - Modèle mémoire distribuée
  - Modèle hybride
  - Méthodologie
- 3 Présentation du code HYDRO**
  - Description
  - Calcul de la solution
  - Version Hybride du code HYDRO
- 4 Outils de débogage HPC**
  - DDT
- 5 Travaux pratiques**
  - Environnement et mise en oeuvre du code HYDRO
  - TP
  - Exercice 1
  - Exercice 2
  - Exercice 3
  - Exercice 4
  - Exercice 5
- 6 Conclusion**

- 1 **Introduction au débogage parallèle**
- 2 **Quelques principes de débogage parallèle**
  - Modèle mémoire partagée
  - Modèle mémoire distribuée
  - Modèle hybride
  - Méthodologie
- 3 **Présentation du code HYDRO**
  - Description
  - Calcul de la solution
  - Version Hybride du code HYDRO
- 4 **Outils de débogage HPC**
  - DDT
- 5 **Travaux pratiques**
  - Environnement et mise en oeuvre du code HYDRO
  - TP
  - Exercice 1
  - Exercice 2
  - Exercice 3
  - Exercice 4
  - Exercice 5
- 6 **Conclusion**

## Problématiques liées au débogage d'applications parallèles HPC

- blocages,
- accès concurrents à la mémoire,
- apparition d'erreurs à partir d'un grand nombre de processus,
- apparition d'erreurs après un temps de calcul conséquent,
- différences des résultats dues à l'ordre d'exécution des calculs (calcul arithmétique en particulier),
- etc...

## Les débogueurs parallèles ne peuvent pas tout résoudre malgré des fonctionnalités très puissantes

- visualisations des valeurs de variables sur l'ensemble des tâches,
- synchronisation ou non des tâches ou des processus,
- attachement du débogueur à tous les processus en cours d'exécution,
- etc...

**Ce document présente un ensemble de scénarios pour lesquels l'utilisation de débogueurs parallèles facilite la correction d'erreurs.**

- Nous vous présenterons le débogueur parallèle :
  - **Linaro Forge DDT**
- Les travaux pratiques porteront sur le code de simulation HYDRO, code parallèle hybride (MPI, OpenMP) représentatif des méthodes de décomposition de domaine.  
Dans ce code ont été introduites des erreurs exclusivement liées à ses aspects parallèles.

- 1 Introduction au débogage parallèle
- 2 **Quelques principes de débogage parallèle**
  - Modèle mémoire partagée
  - Modèle mémoire distribuée
  - Modèle hybride
  - Méthodologie
- 3 **Présentation du code HYDRO**
  - Description
  - Calcul de la solution
  - Version Hybride du code HYDRO
- 4 **Outils de débogage HPC**
  - DDT
- 5 **Travaux pratiques**
  - Environnement et mise en oeuvre du code HYDRO
  - TP
  - Exercice 1
  - Exercice 2
  - Exercice 3
  - Exercice 4
  - Exercice 5
- 6 **Conclusion**

## Rappel :

Dans le modèle de programmation à mémoire partagée, les données peuvent être stockées :

- soit dans la mémoire partagée accessible par tous les *threads* (**variables partagées**),
- soit dans un espace mémoire (pile), local à chacun des *threads* (**variables privées**).

## Documentation :

Voir le cours OpenMP de l'IDRIS (<http://www.idris.fr/formations/openmp/>).



## Erreurs possibles : mauvais choix du statut explicite des variables

- Mauvais choix du statut explicite des variables (partagées et privées).
- Absence de protection des données partagées (oubli d'une **section critique**). Dans cet exemple, le programme donne des résultats différents lors de chaque exécution.

```
program protec
USE OMP_LIB
implicit none
integer :: p, nthreads
p=1
!$OMP PARALLEL
!$OMP SINGLE
print *, 'nthreads=', OMP_GET_NUM_THREADS()
!$OMP END SINGLE
p = p * 2
!$OMP END PARALLEL
print *, "p=", p
end program protec
```

examples/protec.f90

```
$ export OMP_NUM_THREADS=8
$ ./protec
nthreads= 8
p= 16
$ ./protec
nthreads= 8
p= 64
```

```
program protec_solution
USE OMP_LIB
implicit none
integer :: p, nthreads
p=1
!$OMP PARALLEL
!$OMP SINGLE
print *, 'nthreads=', OMP_GET_NUM_THREADS()
!$OMP END SINGLE
!$OMP CRITICAL
p = p * 2
!$OMP END CRITICAL
!$OMP END PARALLEL
print *, "p=", p
end program protec_solution
```

examples/protec\_solution.f90

```
$ export OMP_NUM_THREADS=8
$ ./protec_solution
nthreads= 8
p= 256
```

## Erreurs possibles : absence de barrière de synchronisation

- Absence de barrière de synchronisation par exemple entre l'utilisation et la modification d'une variable partagée dans une région parallèle pouvant engendrer un blocage ou des résultats faux.

```
program synchro
USE OMP_LIB
implicit none
integer ,dimension(10) :: a
integer :: i,n,it ,nthreads ,iter=5
n=1
!$OMP PARALLEL
!$OMP SINGLE
print *, 'nthreads=' ,OMP_GET_NUM_THREADS()
!$OMP END SINGLE
do it=1,iter
!$OMP DO
do i=1,10
a(i)=n
enddo
!$OMP END DO
n=sum(a)
enddo
!$OMP END PARALLEL
print *, n
end program synchro
```

examples/synchro.f90

```
$ export OMP_NUM_THREADS=8
$ ./synchro
nthreads= 8
9423442
```

## Erreurs possibles : absence de barrière de synchronisation

```
program synchro_solution
USE OMP_LIB
implicit none
integer ,dimension(10) :: a
integer :: i,n,it ,nthreads ,iter=5
n=1
!$OMP PARALLEL
!$OMP SINGLE
print *, 'nthreads=' ,OMP_GET_NUM_THREADS()
!$OMP END SINGLE
do it=1,iter
!$OMP DO
do i=1,10
a(i)=n
enddo
!$OMP END DO
n=sum(a)
!$OMP BARRIER
enddo
!$OMP END PARALLEL
print *, n
end program synchro_solution
```

examples/synchro\_solution.f90

```
$ export OMP_NUM_THREADS=8
$ ./synchro_solution
nthreads= 8
100000
```

## Erreurs possibles : allocation d'une zone privée dans une région parallèle

- Allocation d'une zone privée dans une région parallèle : un problème de dépassement mémoire peut survenir à partir d'un certain nombre de *threads*.

```
program allocation
  USE OMP_LIB
  implicit none
  integer :: nb_taches,somme,i,nt,taille_tableau=40000000
  integer,dimension(:),allocatable :: tableau
  !$OMP PARALLEL PRIVATE(nb_taches, tableau, somme)
  nb_taches = OMP_GET_NUM_THREADS()
  !$OMP SINGLE
  do nt=1,nb_taches
    !$OMP TASK
    allocate(tableau(taille_tableau))
    tableau=0
    do i=1,taille_tableau,100
      tableau(i)=nt
    enddo
    somme=SUM(tableau)
    print *,somme
    deallocate(tableau)
    !$OMP END TASK
  enddo
  !$OMP END SINGLE
  !$OMP END PARALLEL
end program allocation
```

examples/allocation.f90

## Erreurs possibles : allocation d'une zone privée dans une région parallèle

```
$ export OMP_NUM_THREADS=1
$ ./allocation
 4000000

$ export OMP_NUM_THREADS=2
$ ./allocation
 4000000
 8000000

$ export OMP_NUM_THREADS=4
$ ./allocation
forrtl: severe (41): insufficient virtual memory
```

## Erreurs dues aux fonctionnalités OpenMP

- Attention au statut implicite d'une variable. Une variable déclarée dans un sous-programme est par défaut privée alors qu'elle est partagée si on l'initialise lors de sa déclaration. Plus généralement, toute variable statique est partagée. Dans cet exemple, le programme donne des résultats faux non déterministes.

```
program implicite
  USE OMP_LIB
  implicit none
  !$OMP PARALLEL
  call sub()
  !$OMP END PARALLEL
end program implicite
```

```
subroutine sub()
  USE OMP_LIB
  implicit none
  integer :: a=92000
  a = a + OMP_GET_THREAD_NUM()
  print *, "a=", a, OMP_GET_THREAD_NUM()
end subroutine sub
```

examples/implicite.f90

```
$ export OMP_NUM_THREADS=4
$ ./implicite
a= 92006 3
a= 92003 1
a= 92002 2
a= 92000 0

$ ./implicite
a= 92000 0
a= 92003 3
a= 92004 1
a= 92006 2
```

```
program implicite_solution
  USE OMP_LIB
  implicit none
  !$OMP PARALLEL
  call sub()
  !$OMP END PARALLEL
end program implicite_solution

subroutine sub()
  USE OMP_LIB
  implicit none
  integer :: a
  a = 92000
  a = a + OMP_GET_THREAD_NUM()
  print *, "a=", a, OMP_GET_THREAD_NUM()
end subroutine sub
```

examples/implicite\_solution.f90

```
$ export OMP_NUM_THREADS=4
$ ./implicite_solution
a= 92003 3
a= 92002 2
a= 92000 0
a= 92001 1
```

## Erreurs dues aux fonctionnalités OpenMP

- Attention aux fonctionnalités implicites des directives OpenMP. C'est le cas de la directive `OMP_END_SINGLE` qui implique une synchronisation de tous les *threads*, à l'inverse de la directive `OMP_END_MASTER`. Le programme suivant donne des résultats faux et indéterministes avec `OMP_END_MASTER` et justes avec `OMP_END_SINGLE`.

```
program master
USE OMP_LIB
implicit none
real, allocatable, dimension(:) :: a,b
integer :: n=10,i
!$OMP PARALLEL
!$OMP MASTER
allocate(a(n),b(n))
read(9,*) a(1:n)
!$OMP END MASTER
!$OMP DO
do i=1,n
    b(i)=10*a(i)
end do
!$OMP END DO
!$OMP END PARALLEL
print *, "b=",b
end program master
```

examples/master.f90

```
$ more fort.9
1.000000000 2.000000000 3.000000000
4.000000000 5.000000000 6.000000000
7.000000000 8.000000000 9.000000000
10.000000000

$ export OMP_NUM_THREADS=1
$ ./master
b= 10.00000000 20.00000000 30.00000000
   40.00000000 50.00000000 60.00000000
   70.00000000 80.00000000 90.00000000
  100.00000000

$ export OMP_NUM_THREADS=2
$ ./master
b= 10.00000000 20.00000000 30.00000000
   40.00000000 50.00000000
   0.1233142649E-41 0.7146622168E-42
   0.1233142649E-41 0.7286752014E-42
   0.1233142649E-41
```



## Erreurs dues aux fonctionnalités OpenMP

```
program master_solution
USE OMP_LIB
implicit none
real, allocatable, dimension(:) :: a,b
integer :: n=10,i
!$OMP PARALLEL
!$OMP SINGLE
allocate(a(n),b(n))
read(9,*) a(1:n)
!$OMP END SINGLE
!$OMP DO
do i=1,n
  b(i)=10*a(i)
end do
!$OMP END DO
!$OMP END PARALLEL
print *, "b=",b
end program master_solution
```

examples/master\_solution.f90

```
$ export OMP_NUM_THREADS=1
$ ./master_solution
b= 10.00000000 20.00000000 30.00000000
   40.00000000 50.00000000 60.00000000
   70.00000000 80.00000000 90.00000000
   100.00000000

$ export OMP_NUM_THREADS=2
$ ./master_solution
b= 10.00000000 20.00000000 30.00000000
   40.00000000 50.00000000 60.00000000
   70.00000000 80.00000000 90.00000000
   100.00000000
```

## Rappel :

- Dans le modèle de programmation à mémoire distribuée, chaque processus a accès uniquement à sa propre mémoire.
- Pour accéder à des données stockées dans la mémoire d'autres processus, il est nécessaire d'échanger des messages avec ces processus, par des appels à des sous-programmes de communication.

## Documentation :

Voir le cours MPI de l'IDRIS (<http://www.idris.fr/formations/mpl/>).

## Erreurs possibles :

- Erreurs dans les arguments des sous-programmes de communication (adresse, taille, type des messages) : les valeurs reçues sont différentes de celles envoyées d'où la possibilité de résultats faux.

```
integer , parameter      :: nb=100
integer , dimension(nb) :: valeurs
.....
if (rang == 2) then
  call MPI_SEND(valeurs(1),nb,MPI_INTEGER,5,100,MPI_COMM_WORLD,code)
elseif (rang == 5) then
  call MPI_RECV(valeurs(1),nb,MPI_DOUBLE_PRECISION,2,100,MPI_COMM_WORLD,statut,code)
end if
```

- Mauvaise gestion des étiquettes dans les communications point à point : blocages.

```
integer , parameter      :: nb=100
integer , dimension(nb) :: champs
.....
if (rang == 2) then
  call MPI_SEND(champs(1),nb,MPI_INTEGER,5,100,MPI_COMM_WORLD,code)
elseif (rang == 5) then
  call MPI_RECV(champs(1),nb,MPI_INTEGER,2,200,MPI_COMM_WORLD,statut,code)
end if
```

## Erreurs possibles :

- Les communications point à point (*MPI\_SEND*) sont bloquantes.  
Suivant l'implémentation de MPI, cette fonction passe du mode bufferisé (*MPI\_BSEND*) au mode synchrone (*MPI\_SSEND*) à partir d'une certaine taille de message.  
Un code peut ainsi bloquer après avoir augmenté la taille du problème.

```
! On suppose avoir exactement 2 processus
```

```
num_proc=mod(rang+1,2)
```

```
call MPI_SEND(rang+1000,1,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD,code)
```

```
call MPI_RECV(valeur,1,MPI_INTEGER,num_proc,etiquette,MPI_COMM_WORLD,&  
MPI_STATUS_IGNORE,code)
```

## Erreurs possibles :

- Mauvaise gestion des communications non-bloquantes avec *MPI\_ISEND* et *MPI\_IRECV*, ici on ne s'assure jamais que toutes les communications sont bien terminées (avec la fonction *MPI\_WAITALL*) avant de les utiliser à nouveau : les valeurs envoyées sont différentes de celles reçues.

```
! Envoi au voisin Est et reception du voisin Ouest
CALL MPI_IRECV(u(,), 1, type_colonne, voisin(W), &
             etiquette, comm2d, requete(1), code)
CALL MPI_ISEND(u(,), 1, type_colonne, voisin(E), &
              etiquette, comm2d, requete(2), code)
! Envoi au voisin Ouest et reception du voisin Est
CALL MPI_IRECV(u(,), 1, type_colonne, voisin(E), &
             etiquette, comm2d, requete(3), code)
CALL MPI_ISEND(u(,), 1, type_colonne, voisin(W), &
              etiquette, comm2d, requete(4), code)
```

## Erreurs possibles :

- Dépassements de valeurs entières liés à la taille des cas tests traités en parallèle.

```
if (rang==0) then
  print *, ' MPI Execution with ',nb_procs, ' processes ( ',dims(1), 'x',dims(2), ' )'
  print *, ' and ',nthreads, ' thread by process'
  print *, ' Starting time integration, nx = ',nx, ' ny = ',ny
  print *, 'Global size of the domain nx*dims(1)*ny*dims(2) ',nx*dims(1)*ny*dims(2)
  allocate (u_global(nx*dims(1)*ny*dims(2)), STAT=AllocateStatus)
  if (AllocateStatus /= 0) then
    print *, "*** Not enough memory *** : ",nx*dims(1)*ny*dims(2)
    STOP
  end if
  u_global(1)=nx*dims(1)
end if
```

```
MPI Execution with          512  processes (          32 x          16 )
and          2  thread by process
Starting time integration, nx =          2000  ny =          4000
Global size of the domain nx*dims(1)*ny*dims(2)  -198967296
fortrtl: severe (174): SIGSEGV, segmentation fault occurred
```

## Rappel :

- Chaque processus MPI exécute plusieurs *threads* OpenMP.
- La bibliothèque MPI dispose d'un mécanisme ***MPI\_INIT\_THREAD*** permettant de choisir le niveau de support pour la gestion des appels MPI par les *threads* OpenMP.

```
required = MPI_THREAD_SERIALIZED
call MPI_INIT_THREAD(required , provided , code)
if (nthreads > 1 .and. provided < required) then
  print *, 'Multithreading level too small'
endif
```

- Le niveau de support fourni par cette fonction :
  - peut, selon l'implémentation MPI, ne pas être identique au niveau demandé,
  - impose des restrictions dans la gestion des appels MPI par les *threads* OpenMP.

## Documentation :

Voir le cours Hybride de l'IDRIS (<http://www.idris.fr/formations/hybride/>).

## Erreurs possibles :

- Utilisation de *MPI\_INIT* au lieu de *MPI\_INIT\_THREAD*.
- Ne pas vérifier le niveau de support fourni.
- Non respect des restrictions imposées par le niveau de support obtenu, par exemple, en mode *MPI\_THREAD\_SERIALIZED* :
  - tous les *threads* peuvent faire des appels MPI mais un seul à la fois ;
  - l'erreur est de ne pas protéger ces appels.



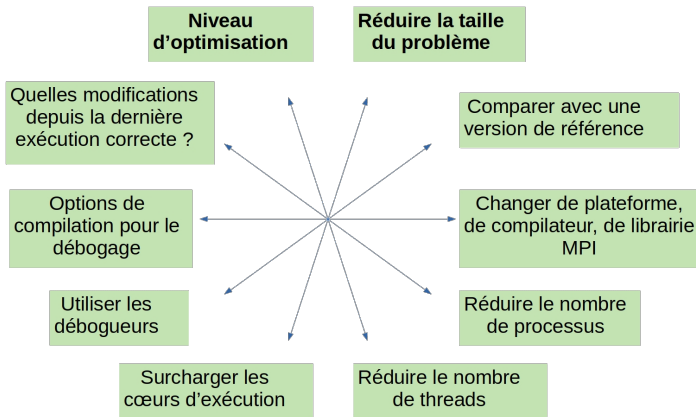


Figure 1 – Schéma de débogage

- 1 Introduction au débogage parallèle
- 2 Quelques principes de débogage parallèle
  - Modèle mémoire partagée
  - Modèle mémoire distribuée
  - Modèle hybride
  - Méthodologie
- 3 **Présentation du code HYDRO**
  - Description
  - Calcul de la solution
  - Version Hybride du code HYDRO
- 4 Outils de débogage HPC
  - DDT
- 5 Travaux pratiques
  - Environnement et mise en oeuvre du code HYDRO
  - TP
  - Exercice 1
  - Exercice 2
  - Exercice 3
  - Exercice 4
  - Exercice 5
- 6 Conclusion

- HYDRO est une version simplifiée du code d'astrophysique RAMSES.
- Code de mécanique des fluides (CFD), qui résout les équations d'Euler compressibles de l'hydrodynamique en 2D.
- Méthode volumes finis utilisant un schéma de Godunov d'ordre 2, avec résolution d'un problème de Riemann (calcul du flux numérique) à chaque interface sur une grille régulière cartésienne 2D.
- 1500 lignes pour la version séquentielle F90.

- A chaque pas de temps, le domaine discrétisé est stocké dans le tableau `uold(1:nx+4,1:ny+4,1:nvar)`.
- Les éléments `uold(i,j,1:nvar)` de ce tableau sont calculés à partir des éléments :
  - `uold(i-2,j,1:nvar)`, `uold(i-1,j,1:nvar)`,  
`uold(i+1,j,1:nvar)`, `uold(i+2,j,1:nvar)`,
  - `uold(i,j-2,1:nvar)`, `uold(i,j-1,1:nvar)`,  
`uold(i,j+1,1:nvar)`, `uold(i,j+2,1:nvar)`.

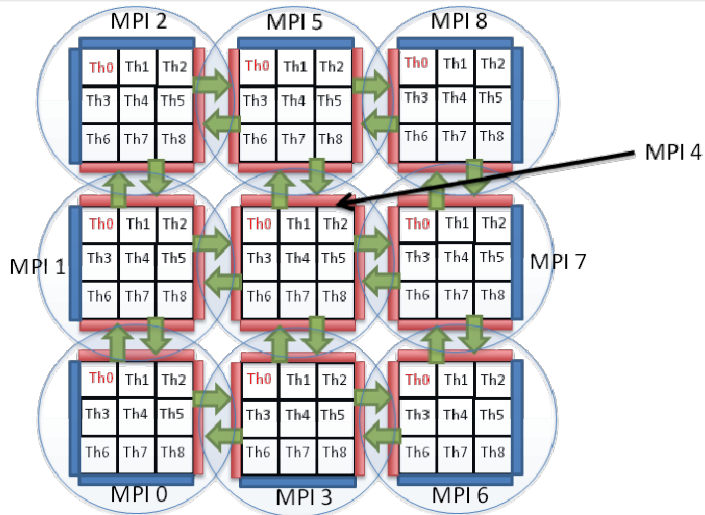
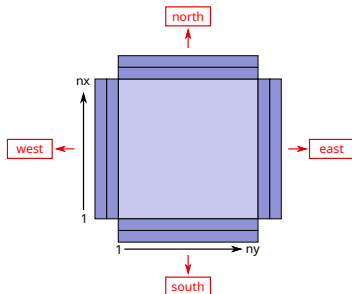


Figure 2 – Version Hybride du code HYDRO

- Utilisation d'une topologie MPI 2D.
- Détermination des voisins Nord, Sud, Est, Ouest pour chaque processus MPI.
- Création de 4 zones "fantôme" pour chaque sous-domaine composées de 2 lignes pour le Nord et le Sud et de 2 colonnes pour l'Est et l'Ouest.
- Création de deux types dérivés composés respectivement de 2 lignes et 2 colonnes.
- Echange d'éléments de ces 2 types entre les processus voisins en utilisant les zones "fantôme".
- Utilisation d'une réduction pour calculer un pas de temps global.



- 1 **Introduction au débogage parallèle**
- 2 **Quelques principes de débogage parallèle**
  - Modèle mémoire partagée
  - Modèle mémoire distribuée
  - Modèle hybride
  - Méthodologie
- 3 **Présentation du code HYDRO**
  - Description
  - Calcul de la solution
  - Version Hybride du code HYDRO
- 4 **Outils de débogage HPC**
  - DDT
- 5 **Travaux pratiques**
  - Environnement et mise en oeuvre du code HYDRO
  - TP
  - Exercice 1
  - Exercice 2
  - Exercice 3
  - Exercice 4
  - Exercice 5
- 6 **Conclusion**

Il existe actuellement deux débogueurs HPC sous licence :

- *DDT* (Linaro, précédemment Arm)

<https://www.linaroforge.com>

Disponible sur Jean Zay

- *TotalView* (Rogue Wave Software)

<https://help.totalview.io/current/HTML/index.html>

Indisponible sur Jean Zay



DDT permet de déboguer des codes séquentiels et parallèles.

- Langages supportés : C, C++, Fortran, Python
- Paradigmes de programmation parallèle :
  - CPU : MPI, OpenMP
  - GPU : CUDA C/C++/Fortran, OpenACC
  - Codes hybrides : MPI + OpenMP ou MPI + CUDA/OpenACC
- Applications MPMD (Mutiple Program Multiple Data)

- Points d'arrêt, points d'arrêt conditionnels
- Affichage de la pile d'appel
- Affichage des valeurs, des tableaux, par processus MPI, par thread
- Changement des valeurs sans recompilation
- Watchpoints : arrêt sur un changement d'une variable
- Débogage de mémoire (guard bytes, allocations)

DDT est exécuté sur les nœuds de calcul.

L'**interface graphique** (GUI) est disponible :

- soit sur les nœuds frontaux
- soit installée localement

**Modes d'utilisation** de DDT sur des travaux batch :

- **Mode interactive** : le travail batch est lancé depuis l'interface graphique
- **Reverse Connect** : le travail batch se connecte au GUI (le seul mode possible avec Remote Client)
- **Attachement** sur un code en exécution (pratique en cas de blocage du code)
- **Exécution offline** : sans utilisation de GUI (peut être utile pour le débogage de la mémoire)

Le mode *Reverse Connect* est celui conseillé sur Jean Zay.

Permet d'éviter l'affichage graphique X Window au travers de la connexion ssh.

## Installation du Remote Client

- Peut être téléchargé gratuitement sur le site de Linaro :  
<https://www.linaroforge.com/downloadForge/>
- La version doit correspondre à la version disponible sur un supercalculateur.

Sur Jean Zay :

```
$ module load linaro-forge
$ ddt --version
```

## Configuration du Remote Client

- Lancement du GUI Linaro Forge (\$ ddt &)
- Remote Launch -> Configure -> Add
  - Host Name (rebonds multiples possibles)
  - Remote Installation Directory

Sur Jean Zay :

```
$ module load linaro-forge
$ module show linaro-forge
...
prepend-path PATH /gpfslocalsys/arm/forge/23.0.4/bin
```

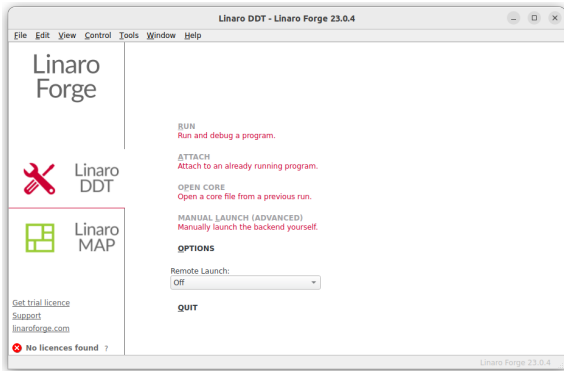


Figure 3 – Fenêtre de démarrage

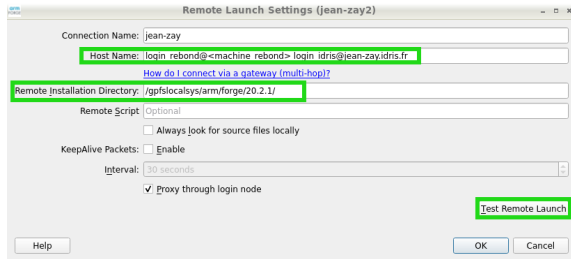


Figure 4 – Remote Launch

# DDT : Lancement du travail via Reverse Connect

## Compilation du code

- Fortran,C++ : les options **-O0 -g**
- CUDA : les options de nvcc **-O0 -g -G**

## Lancement de l'interface graphique sur Jean Zay

```
rm -rf $HOME/.allinea
module load linaro-forge
ddt&
```

## Edition du script de soumission

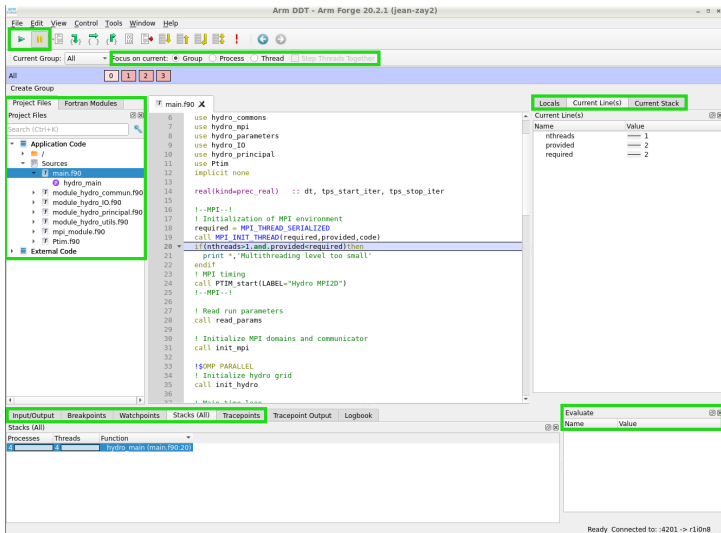
```
# Ajout du module linaro-forge pour DDT
module load linaro-forge
...
# Execution du code via DDT en mode Reverse Connect.
ddt --connect srun ./exec_to_debug
```

## Soumission du travail

```
$ sbatch votre_script.slurm
```

Quand votre travail sera démarré, DDT affichera une demande de connexion.

## Exemple MPI/OpenMP : 4 processus MPI, 2 threads



- Code source (modifiable)
- Fichiers de projet (Rechercher)
- Contrôles de processus (Run, Pause, ...)
- Groupes de processus (processus MPI, threads)
- Fenêtre Évaluer (tracer ou même changer une valeur)
- I/O, Breakpoints, Watchpoints, Stack

Figure 5 – Fenêtre principale

The screenshot shows the Arm DDT - Arm Forge 20.2.1 (jean-zay2) IDE. The main editor displays Fortran code in `main.f90`. Line 41 is highlighted, containing the statement `nstep=nstep+1`. A red circle is placed over this line. The 'Current Line(s)' panel on the right shows the variable `nstep` with a value of 4. The 'Breakpoints' table at the bottom shows the configuration of the conditional breakpoint.

Processes	Threads	File	Line	Function	Condition	Start After	Trigger Every	Stop After	Full path
<input checked="" type="checkbox"/>	All	all	main.f90	L_MAIN__33_par_region0_2_0	nstep==4	0	1	Forever	/gpfsdswork/...

- positionner le point d'arrêt désiré,
- cliquer avec le bouton droit sur le point d'arrêt,
- sélectionner *Edit breakpoint for All*,
- sélectionner le langage du programme (*C, Fortran*),
- dans *Condition* formuler la condition d'arrêt

Figure 6 – Point d'arrêt conditionnel



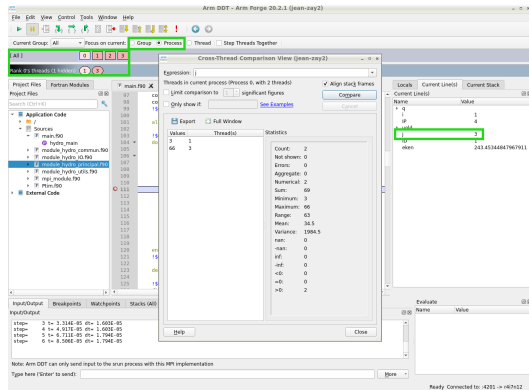


Figure 7 – Valeur d'une variable sur chaque *thread*

Array Expression: `uold($i, $j, $k)` Evaluate

Distributed Array Dimensions: None [How do I view distributed arrays?](#)

Staggered Array [What does this do?](#)  Align Stack Frames

Auto-update

Range of \$i: From: 1 To: 129 Display: Rows

Range of \$j: From: 1 To: 129 Display: Rows

Range of \$k: From: 1 To: 4 Display: Columns

Only show if: [See Examples](#)

Data Table Statistics

[Goto](#) [Visualize](#) [Export](#) [Full Window](#)

		k			
		1	2	3	4
j 1	i 1	0	0	0	0
	2	0	0	0	0
	3	0.31438407631845672	5.3714663897057378	-16.801231389204212	7050.2206370864478
	4	0.71781261716487621	32.798673500534861	-33.479107887017165	8727.247427895727
	5	1.4860597897931622	43.617149111203496	-29.809821118631852	4857.2108646915294
	6	1.1374746349389422	3.3212286165107483	-2.8409029113032109	165.5401888282006
	7	1.0035112812542462	0.0042173263120142638	-0.0023428107411552282	0.0096583146623512601
	8	1.0000008255120985	1.9382305332138328e-09	-3.1530773545048547e-10	1.000001326227816e-05

Help Close

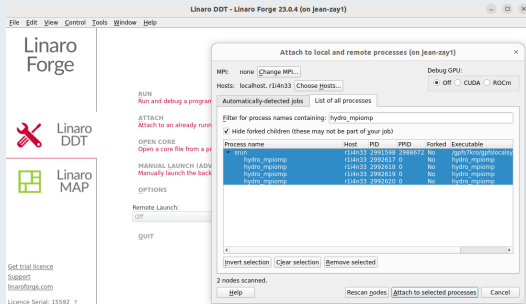
Figure 8 – Visualisation d'un tableau

## Attachement sur un code en exécution

- soumettre l'exécution d'un code en mode batch,
- repérer le noeud d'exécution :

```
$ squeue -u $USER
      JOBID PARTITION    NAME    USER  ST       TIME  NODES NODELIST(REASON)
      1270775      cpu_p1    Hydro    user   R         0:18     1  r1i0n33
```

- lancer le débogueur et cliquer sur *Attach to an already running program*,
- cliquer sur le bouton *Choose Hosts* pour ajouter ce noeud,





- 1 **Introduction au débogage parallèle**
- 2 **Quelques principes de débogage parallèle**
  - Modèle mémoire partagée
  - Modèle mémoire distribuée
  - Modèle hybride
  - Méthodologie
- 3 **Présentation du code HYDRO**
  - Description
  - Calcul de la solution
  - Version Hybride du code HYDRO
- 4 **Outils de débogage HPC**
  - DDT
- 5 **Travaux pratiques**
  - Environnement et mise en oeuvre du code HYDRO
  - TP
  - Exercice 1
  - Exercice 2
  - Exercice 3
  - Exercice 4
  - Exercice 5
- 6 **Conclusion**

- Plateforme utilisée :
  - DDT sur la machine (*Jean Zay*, IDRIS), HPE SGI 8600
    - 28800 cœurs
    - 40 cœurs par nœud SMP

- Se placer dans le répertoire

```
$WORK/TP/hydro_mpiopenmp
```

- Sont fournis

- un fichier Makefile pour la compilation,
- un fichier job.slurm pour la soumission en traitement par lots,
- les fichiers sources à corriger en Fortran,
- un répertoire *Solution* contenant les solutions des exercices proposés.

- Chargement de l'environnement : *module load intel-all/2019.4 linaro-forge*

- Compilation : *make* (options de compilation pour le débogage : *-O0 -g*)

- Soumission en mode batch en utilisant le fichier de soumission *job.slurm* :

```
sbatch job.slurm  
avec
```

```
--ntasks=xx          nombre xx de processus MPI  
--cpus-per-task=yy   nombre yy de threads OpenMP
```

- Fichier de référence pour vérifier les résultats numériques *output\_step\_sedov\_250x250.ref*

- Pour compiler le code

```
make
```

- Pour effacer tous les fichiers .o et .mod

```
make clean
```

- Pour soumettre un job

```
sbatch job.slurm
```

- Pour suivre un job

```
squeue -u $USER
```

- Un fichier de retour de job se trouve dans le répertoire de soumission sous le nom

```
Hydro.{Numerodujob}.out
```



Le but des exercices est d'obtenir une version correcte du code avec 4 processus MPI et 2 *threads* OpenMP.

- Tester le code avec 4 processus MPI et 2 *threads* OpenMP en utilisant le fichier de soumission *job.slurm*.
- Que constatez-vous ?

## Exercice 1 : Débogage séquentiel

Nous allons commencer par faire du débogage séquentiel, en exécutant le programme avec 1 processus MPI et 1 *thread* OpenMP.

- Modifier le fichier de soumission *job.slurm*

```
--ntasks=1  
--cpus-per-task=1
```

- Soumettre le job batch.
- Que constatez-vous ?
- Utiliser le débogueur pour localiser cette erreur.
- Corriger l'erreur.
- Vérifier la bonne exécution séquentielle du code (1 processus MPI, 1 *thread* OpenMP).

## Exercice 2 : Débogage en mémoire partagée

- Tester le programme en OpenMP avec 1 processus MPI et 8 *threads* OpenMP.
- Modifier le fichier de soumission en conséquence avant de soumettre le job.
- Que constatez-vous ?
- Exécuter le programme via le débogueur.
- Quand le programme se bloque (les résultats ne défilent plus), stopper le programme et regarder à quel endroit sont arrêtés les *threads*.
- Corriger l'erreur.
- Vérifier que le programme s'exécute maintenant jusqu'au bout.

## Exercice 3 : Débogage en mémoire partagée

- Le code s'exécute maintenant jusqu'au bout mais qu'en est-il des résultats numériques ?
- Vérifier les résultats en OpenMP (1 processus MPI, 4 *threads* OpenMP) qui se trouvent dans votre fichier de retour de job avec le fichier de référence *output\_step\_sedov\_250x250.ref*, ceci sur les premières itérations.
- Que constatez-vous, notamment sur la valeur de la variable *dt* ?
- Utiliser le débogueur pour chercher d'où provient l'erreur.
- Chercher la variable à partir de laquelle est calculé *dt*.
- Chercher l'endroit où elle est calculée et utilisée, et par quels *threads*.
- Corriger l'erreur.
- Vérifier les résultats.

## Exercice 4 : Débogage en mémoire distribuée

- Nous allons maintenant tester le code avec la version pure MPI (4 processus MPI, 1 *thread* OpenMP). Pour cela, modifier à nouveau le fichier de soumission en conséquence avant de soumettre le job.
- Que constatez-vous au niveau des résultats numériques ?
- Utiliser le débogueur avec 4 processus MPI, 1 *thread* OpenMP.
- Une première piste est de vérifier si chaque processus a bien dans son tableau voisin les bons voisins.
- Corriger l'erreur.
- Vérifier les résultats en MPI (4 processus MPI, 1 *thread* OpenMP).
- Vérifier maintenant les résultats avec 4 processus MPI et 2 *threads* OpenMP.

### Exercice 5 : Débogage en mémoire distribuée

- Nous allons tester le programme sur un plus important d'itération 10 000 au lieu de 100 précédemment.
- Recopier le fichier *input\_sedov\_noio\_250x250\_10000.nml* dans *input\_sedov\_noio\_250x250.nml*.
- Vérifier les résultats avec ce nouveau fichier d'entrée sur 8 processus MPI et 1 *thread* OpenMP.
- Des différences numériques apparaissent vers la 3000 ème itération.
- **Il subsiste encore une erreur !!!!**

## Exercice 5 : Débogage en mémoire distribuée (suite)

- Utiliser le débogueur avec 8 processus MPI et 1 *thread* OpenMP.
- Vérifier les communications : les valeurs reçues correspondent-elles aux valeurs envoyées ? Pour cela :
  - Exécuter le programme pour se positionner à l'itération 3000 (*step= 3000*) en utilisant un point d'arrêt conditionnel.
  - Vérifier ensuite que les données échangées entre chaque processus sont correctes.
- Corriger l'erreur.
- Vérifier maintenant les résultats avec 8 processus MPI, 1 *thread* OpenMP, puis avec 2 *threads* OpenMP.

- 1 **Introduction au débogage parallèle**
- 2 **Quelques principes de débogage parallèle**
  - Modèle mémoire partagée
  - Modèle mémoire distribuée
  - Modèle hybride
  - Méthodologie
- 3 **Présentation du code HYDRO**
  - Description
  - Calcul de la solution
  - Version Hybride du code HYDRO
- 4 **Outils de débogage HPC**
  - DDT
- 5 **Travaux pratiques**
  - Environnement et mise en oeuvre du code HYDRO
  - TP
  - Exercice 1
  - Exercice 2
  - Exercice 3
  - Exercice 4
  - Exercice 5
- 6 **Conclusion**



- *"Debugging is twice as hard as writing the code in the first place".*  
Brian Kerningham (1974)
- D'où le conseil :
  - Bien écrire le code.
  - Commenter.
  - Cas test.
  - Porter sur plusieurs plateformes.
  - ...
- **Attention il n'y a pas d'outil miracle.**