Parallel Debugger Utilisation

Isabelle Dupays
Marie Flé

Translated from French by Cynthia Taupin

Parallel Debugger Utilisation – Plan I

Parallel Debugger Utilisation – Plan II

1 – Introduction to parallel debugging

Problems related to parallel HPC debugging applications

- Appearance of errors with a large number of processors
- Appearance of errors after a long computing time
- Blockings
- Result differences due to the execution order of the calculations (sums, in particular)
- Other, ...

Parallel debuggers cannot solve everything despite very powerful functionalities

- Visualisation of variable values for all the tasks
- Synchronisation or not of the tasks or processes
- Attachment of the debugger to all the processes which are running
- Other, ...

1 – Introduction to parallel debugging

This document presents several scenarios for which the use of parallel debuggers facilitates the correction of errors.

- Two parallel debuggers will be presented :
  - **DDT**
  - **TotalView**
- The hands-on exercises focus on the HYDRO simulation code, a hybrid (MPI, OpenMP) parallel code typical of domain decomposition methods.
  Errors were introduced exclusively into the parallel parts of this code.

2 – Some principles of parallel debugging
2.1 – Shared memory model

Remember :

In the shared memory programming model, data can be stored in one of two places :

- In the shared memory, accessible by all the *threads* (**shared variables**)
- In the stack memory space, local to each of the *threads* (**private variables**).

Documentation :

See the IDRIS OpenMP courses (`http://www.idris.fr/formations/openmp/`).

Possible errors :

- Incorrect choice of the explicit status of the variables (shared and private).
- Lack of shared data protection (omitting a **critical section**). In this example, the program gives different results during each execution.

```
program protection
use omp_lib
implicit none
integer :: p
p=1
!$OMP PARALLEL
p = p * 2
!$OMP END PARALLEL
print *, "p=",p
end program protection
```

Correction :

```
!$OMP CRITICAL
p = p * 2
!$OMP END CRITICAL
```

Possible errors :

- Lack of a synchronisation barrier ; for example, between the use and the modification of a shared variable in a parallel region, can cause deadlock or false results.

```
program synchronisation
use omp_lib
implicit none
integer ,dimension(10) :: a
integer                :: i,n,it
integer                :: iteration=5
n=1
!$OMP PARALLEL
do it=1,iteration
!$OMP DO
do i=1,10
  a(i)=n
enddo
!$OMP END DO
n=sum(a)
enddo
!$OMP END PARALLEL
print *, n
end program synchronisation
```

Correction :

```
Replace the line "n=sum(a)"
by
n=sum(a)
!$OMP BARRIER
```

Possible errors :

- Allocation of a private area in a parallel region : A problem of memory overflow can occur when using a certain number of *threads*.

```
use omp_lib
implicit none
integer :: nb_tasks,sum_array,i,nt,array_size=400000000
integer,dimension(:),allocatable  :: array
!$OMP PARALLEL PRIVATE(nb_tasks, array)
nb_tasks = OMP_GET_NUM_THREADS()
!$OMP SINGLE
do nt=1,nb_tasks
  !$OMP TASK
  allocate(array(array_size))
  do i=1,array_size,100
    array(i)=nt
  enddo
  sum_array=SUM(array)
  print *,sum_array
  deallocate(array)
  !$OMP END TASK
enddo
!$OMP END SINGLE
!$OMP END PARALLEL
```

2 – Some principles of parallel debugging
2.1 – Shared memory model

Errors due to OpenMP functionalities

- Pay attention to the implicit status of a variable. A variable declared in a subroutine is private by default but it is shared if it is initialised when it is declared. Generally, every static variable is shared. In this example, the program gives non-deterministic false results.

```
program implicit
use omp_lib
implicit none
!$OMP PARALLEL
call sub()
!$OMP END PARALLEL
end program implicit
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
```

Correction :

```
Replace the line "integer :: a=92000"
by
integer :: a
a=92000
```

2 – Some principles of parallel debugging
2.1 – Shared memory model

Errors due to OpenMP functionalities

- Pay attention to the implicit functionalities of the OpenMP directives. This is the case of the `OMP END SINGLE` directive which implies a synchronisation of all the *threads*, the opposite of the `OMP END MASTER` directive. The following program gives false and non-deterministic results with `OMP END MASTER` and correct results with `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)=.2*a(i)
end do
!$OMP END DO
!$OMP END PARALLEL
print *, "b=",b
end program master
```

2 – Some principles of parallel debugging
2.2 – Distributed memory model

Remember :

- In the distributed memory programming model, each process only has access to its own memory.
- To access data stored in the memory of other processes, it is necessary to exchange messages with these processes by calls to communication subroutines.

Documentation :

See the IDRIS MPI course (`http://www.idris.fr/formations/mpi/`).

2 – Some principles of parallel debugging
2.2 – Distributed memory model

Possible errors :

- Errors in the communication subroutine arguments (address, size, type of message) : The values received are different than those sent, giving the possibility of false results.

```
integer,parameter      :: nb=100
integer,dimension(nb) :: values
.................................
if (rank == 2) then
  call MPI_SEND(values(1),nb,MPI_INTEGER,5,100,MPI_COMM_WORLD,code)
else if (rang == 5) then
  call MPI_RECV(values(1),nb,MPI_DOUBLE_PRECISION,2,100,MPI_COMM_WORLD,status,code)
end if
```

- Unmatched tags in point-to-point communications : deadlock

```
integer,parameter      :: nb=100
integer,dimension(nb) :: fields
.................................
if (rank == 2) then
  call MPI_SEND(fields(1),nb,MPI_INTEGER,5,100,MPI_COMM_WORLD,code)
else if (rank == 5) then
  call MPI_RECV(fields(1),nb,MPI_INTEGER,2,200,MPI_COMM_WORLD,status,code)
end if
```

2 – Some principles of parallel debugging
2.2 – Distributed memory model

Possible errors :

- Point-to-point (MPI_SEND) communications are blocking.
  Depending on the MPI implementation, this function passes from buffered mode (MPI_BSEND) to synchronous mode (MPI_SSEND) when there is a certain message size.
  Therefore, a code can block if the problem size is increased.

```
! Supposing we have exactly 2 processes
proc_number=mod(rank+1,2)
call MPI_SEND(rank+1000,1,MPI_INTEGER,proc_number,etiquette,MPI_COMM_WORLD,code)
call MPI_RECV(value,1,MPI_INTEGER,proc_number,etiquette,MPI_COMM_WORLD,&
              MPI_STATUS_IGNORE,code)
```

2 – Some principles of parallel debugging
2.2 – Distributed memory model

Possible errors :

- Incorrect synchronisation of non-blocking communications with MPI_ISEND and MPI_IRECV. Here there is never a test to be sure that all the communications are completed (with the MPI_WAITALL function) before re-using them : The sent values are different than the received values.

```
! Send to the East neighbour and reception from the West neighbour
CALL MPI_IRECV(u(,), 1, type_row, neighbour(W), &
               tag, comm2d, request(1), code)
CALL MPI_ISEND(u(,), 1, type_row, neighbour(E), &
               tag, comm2d, request(2), code)
! Send to the West neighbour and reception from the East neighbour
CALL MPI_IRECV(u(,),1,type_row,neighbour(E), &
               tag, comm2d, request(3), code)
CALL MPI_ISEND(u(,),1,type_row,neighbour(W), &
               tag, comm2d, request(4), code)
```

- Stacking of derived datatype constructions or of communicators without liberating them : surpassing memory reservations.

2 – Some principles of parallel debugging
2.2 – Distributed memory model

Possible errors :

- Exceeding integer values due to the test case sizes processed in parallel

```
if (rank==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  threads by process
  Starting time integration, nx =         2000  ny =         4000
 Global size of the domain nx*dims(1)*ny*dims(2)   -198967296
forrtl: severe (174): SIGSEGV, segmentation fault occurred
```

2 – Some principles of parallel debugging
2.3 – Hybrid model

Remember :

- Each MPI process executes several OpenMP *threads*.
- The MPI library has a mechanism `MPI_INIT_THREAD` available which allows you to choose the level of support for the management of MPI calls by the OpenMP *threads*.

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

- The level of support provided by this function could :
  - Be different from the requested level, depending on the MPI implementation.
  - Impose restrictions on the management of MPI calls by the OpenMp *threads*.

Documentation :

See the IDRIS Hybrid course (`http://www.idris.fr/formations/hybride/`).

Possible errors :

- Using `MPI_INIT` instead of `MPI_INIT_THREAD`.
- Not verifying the level of support provided.
- Not respecting the restrictions imposed by the obtained level of support. For example, in `MPI_THREAD_SERIALIZED` mode :
  - All the *threads* can make MPI calls but only one at a time. The error is to not protect these calls.

2 – Some principles of parallel debugging
2.4 – Methodology

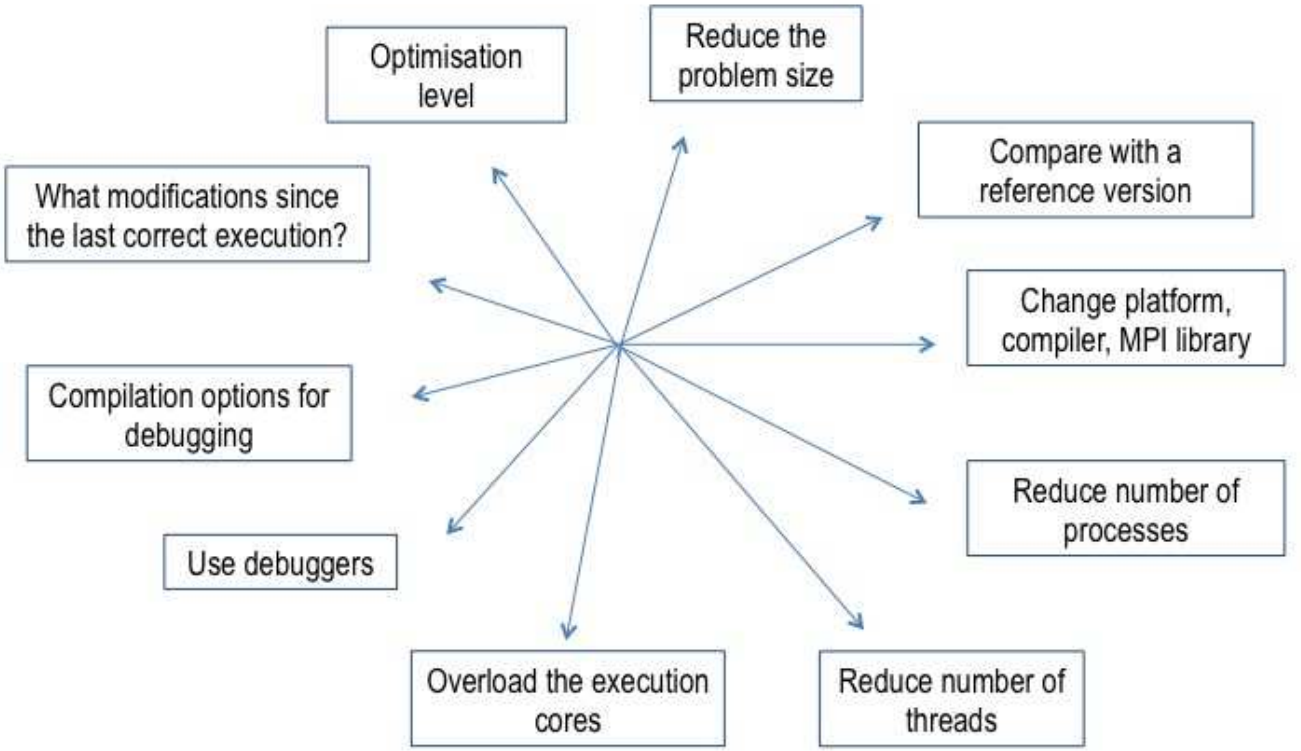


FIGURE 1 – Debugging diagram

3 – Presentation of the HYDRO code
3.1 – Description

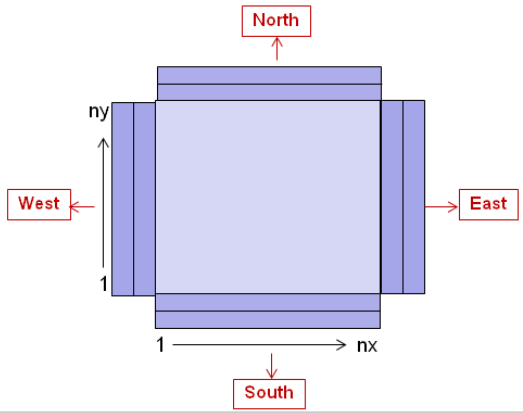
- HYDRO is a simplified version of the RAMSES astrophysics code.
- Computational fluid dynamics (CFD) code, which resolves 2D compressible Euler equations of hydrodynamics.
- Finite volume method using a second-order Godunov scheme with Riemann problem resolution (numerical flux computation) at each interface on a regular 2D cartesian grid.
- 1500 lines for the sequential F90 version.

3 – Presentation of the HYDRO code
3.2 – Calculation of the solution

- At each timestep, the discretized domain is stored in the `uold(1:nx,1:ny,1:nvar)` array.
- The `uold(i,j,1:nvar)` elements of this array are calculated from the following elements :
  - `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)`.

3 – Presentation of the HYDRO code
3.3 – Hybrid version of the HYDRO code

- Use of an MPI 2D topology
- Creation of 4 "phantom" zones for each sub-domain consisting of 2 lines each for the North and the South, and 2 columns each for the East and the West
- Determination of the neighbours to the North, South, East and West for each MPI process
- Creation of two datatypes which are composed, respectively, of 2 lines and 2 columns
- Element exchanges of these 2 datatypes with neighbor processes by using the phantom zones
- Use of a reduction to calculate a global timestep



3 – Presentation of the HYDRO code
3.3 – Hybrid version of the HYDRO code

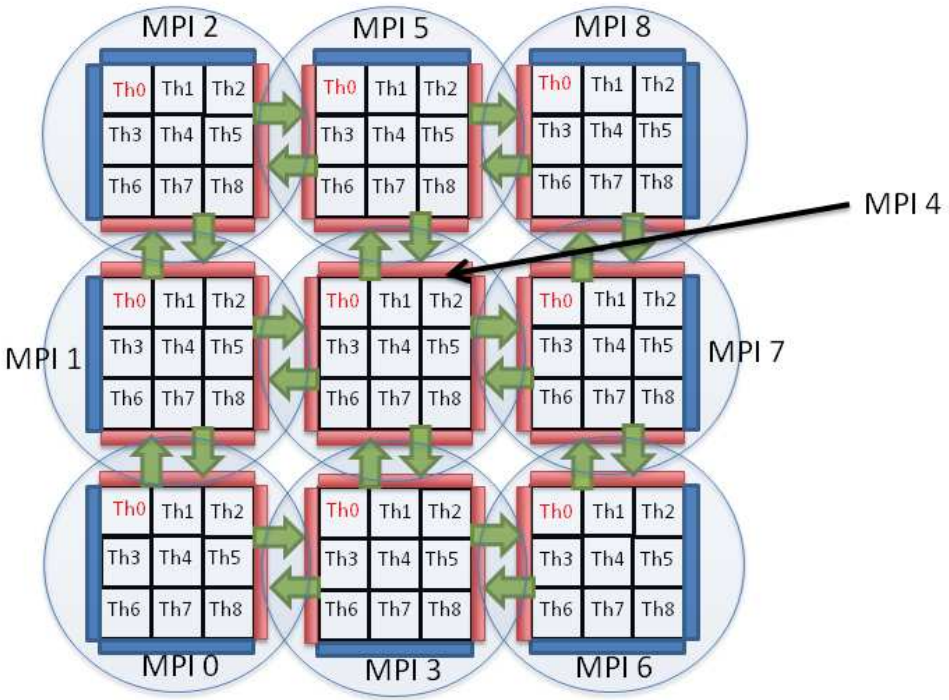


FIGURE 3 – Hybrid version of the HYDRO code

4 – HPC debugging tools
4.1 – Introduction

There are two HPC debuggers which are currently licenced :

- *DDT* (Allinea)
  `http://content.allinea.com/downloads/userguide-forge.pdf`
- *TotalView* (Rogue Wave Software)
  `http://docs.roguewave.com/totalview/8.15.4/pdfs/TotalView_User_Guide.pdf`

DDT allows the debugging of sequential and parallel codes :

- OpenMP
- MPI
- MPI + OpenMP or MPI + CUDA hybrid codes
- Client-server applications

Environment configuration

- *rm -rf ~/.allinea*
- *module load ddt*
- *ddt &*

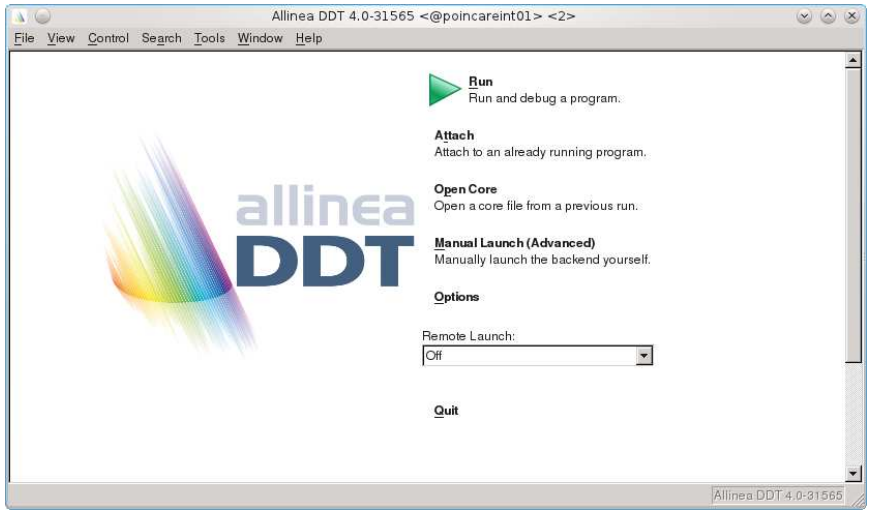


FIGURE 4 – The start window

- Click on *Options*

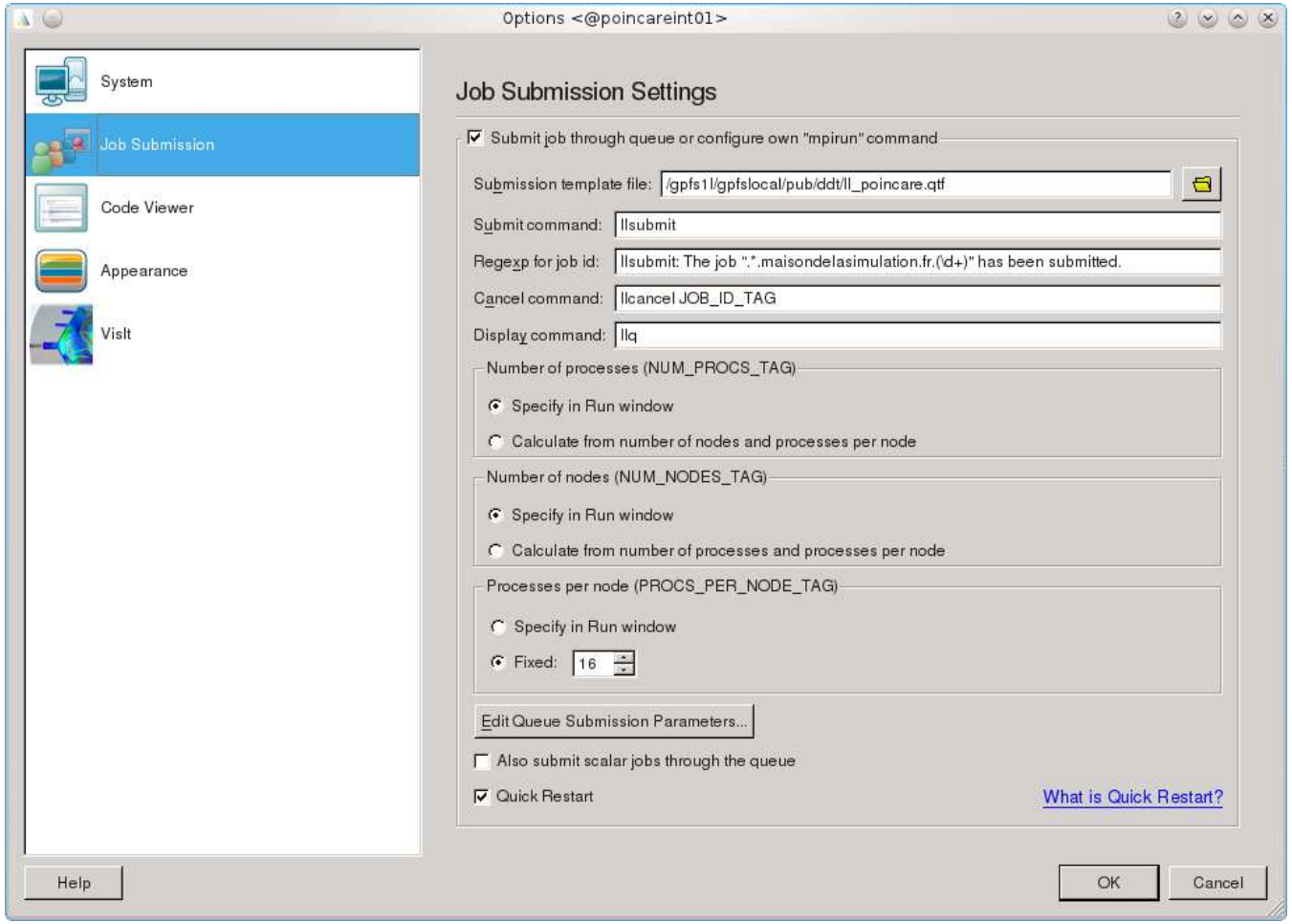


FIGURE 5 – Configuration for job submission

- Click on Job Submission
- Find the name of the .qtf file, job submission model.
- Click on *Ok*, then *Run*

FIGURE 6 – Configuration of the execution

- Complete the fields ! :
  - Application
  - Arguments
  - stdin file
  - Working Directory
  - MPI
    - Number of processes
    - Implementation
    - mpirun arguments
  - OpenMP
    - Number of OpenMP threads
- Click on *Submit*

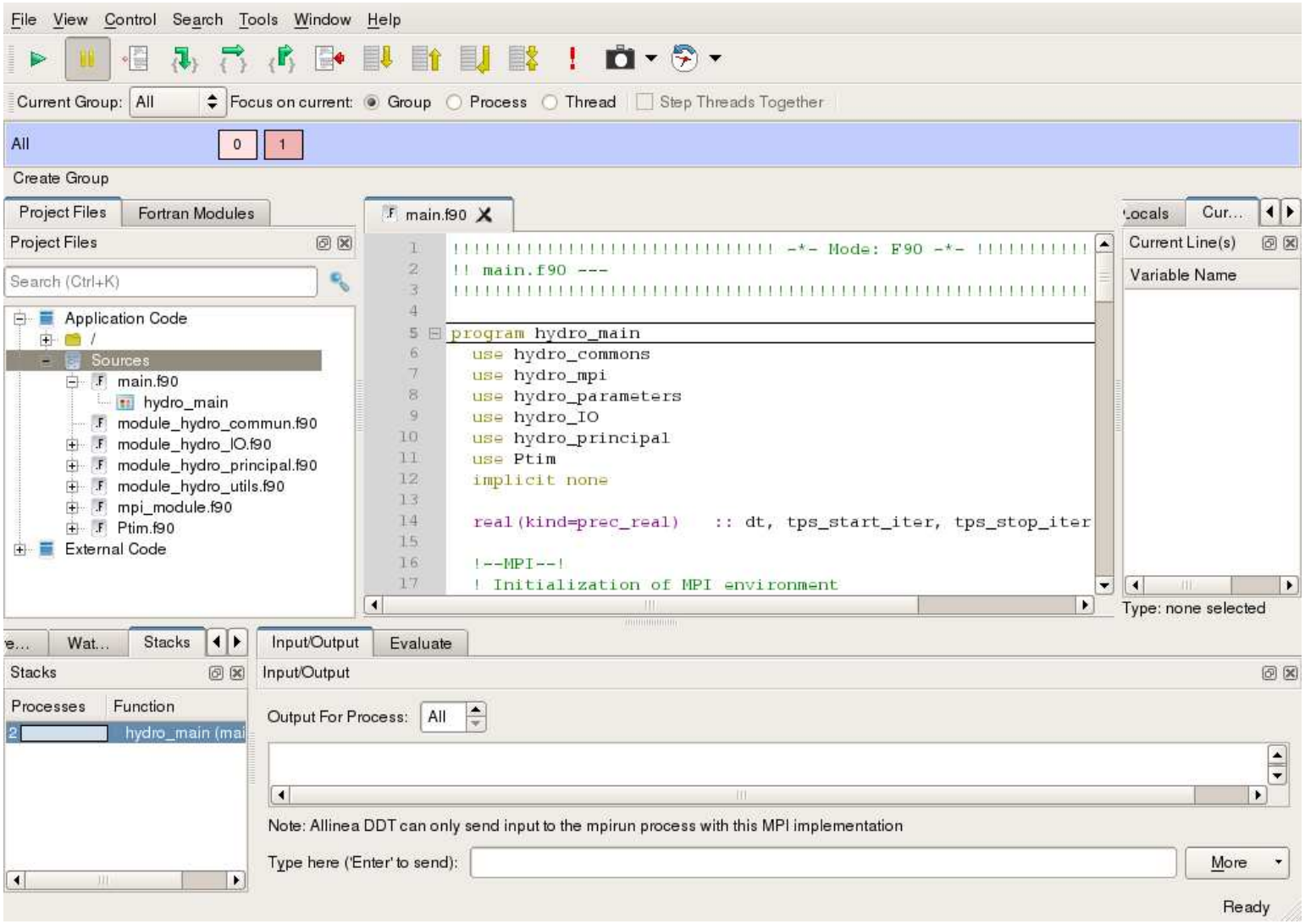


FIGURE 7 – Principal window

- Set a breakpoint and click on the green arrow on the top left

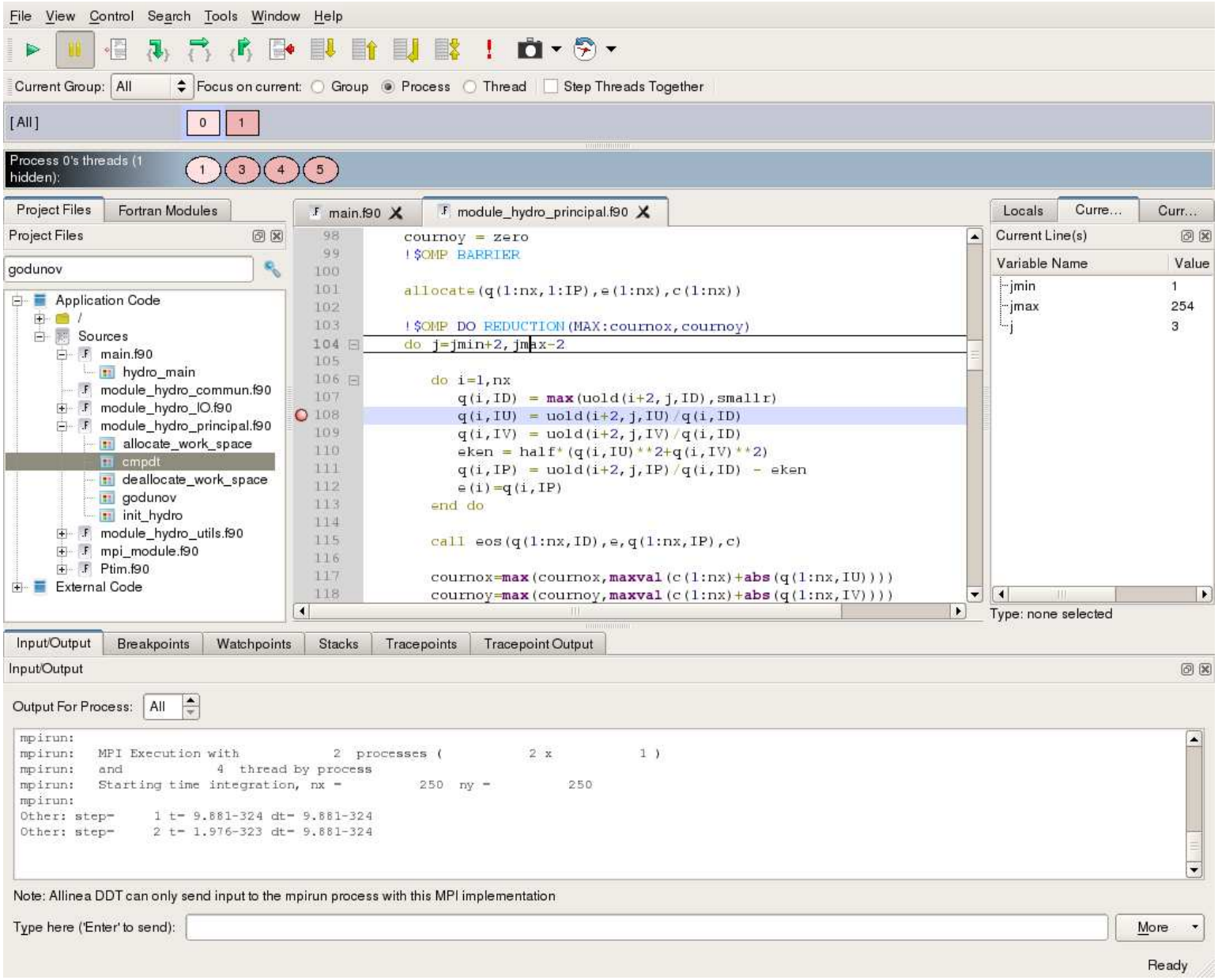


FIGURE 8 – Breakpoint in an OpenMP parallel region

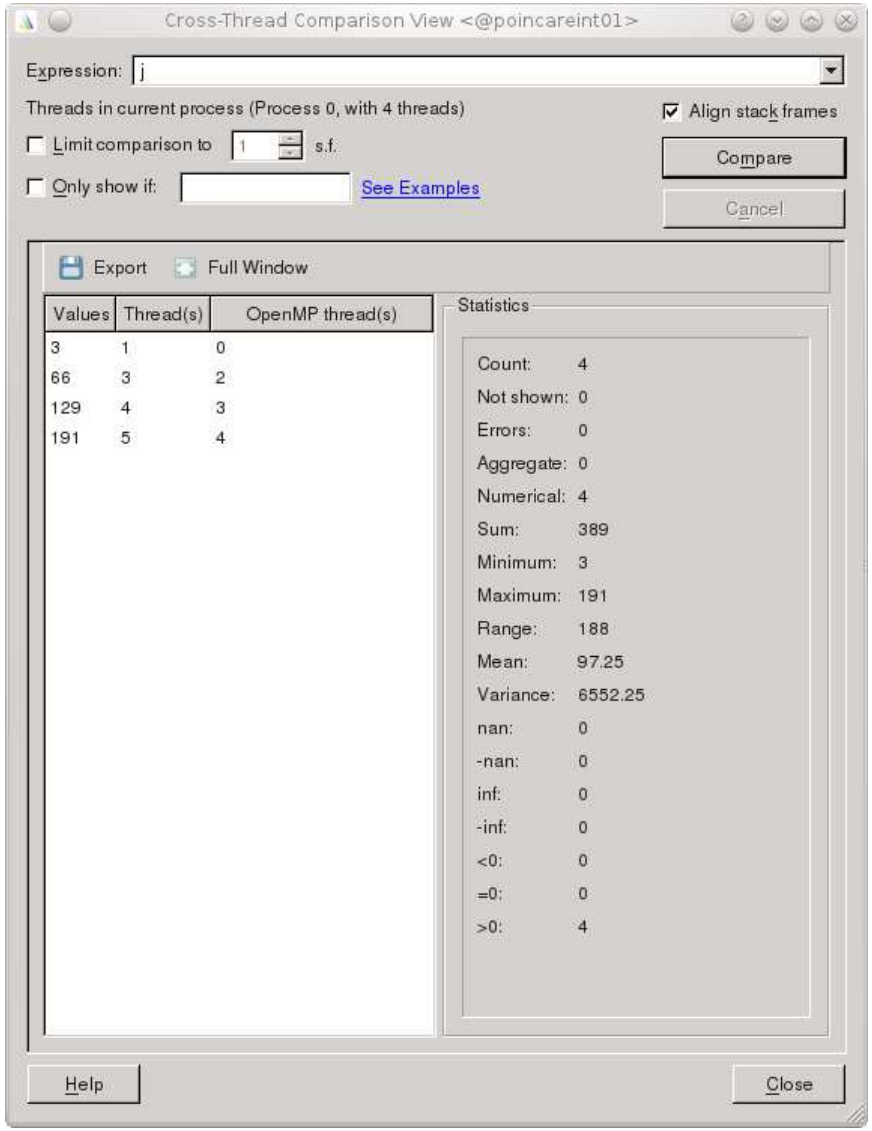


FIGURE 9 – Value of variable on each *thread*

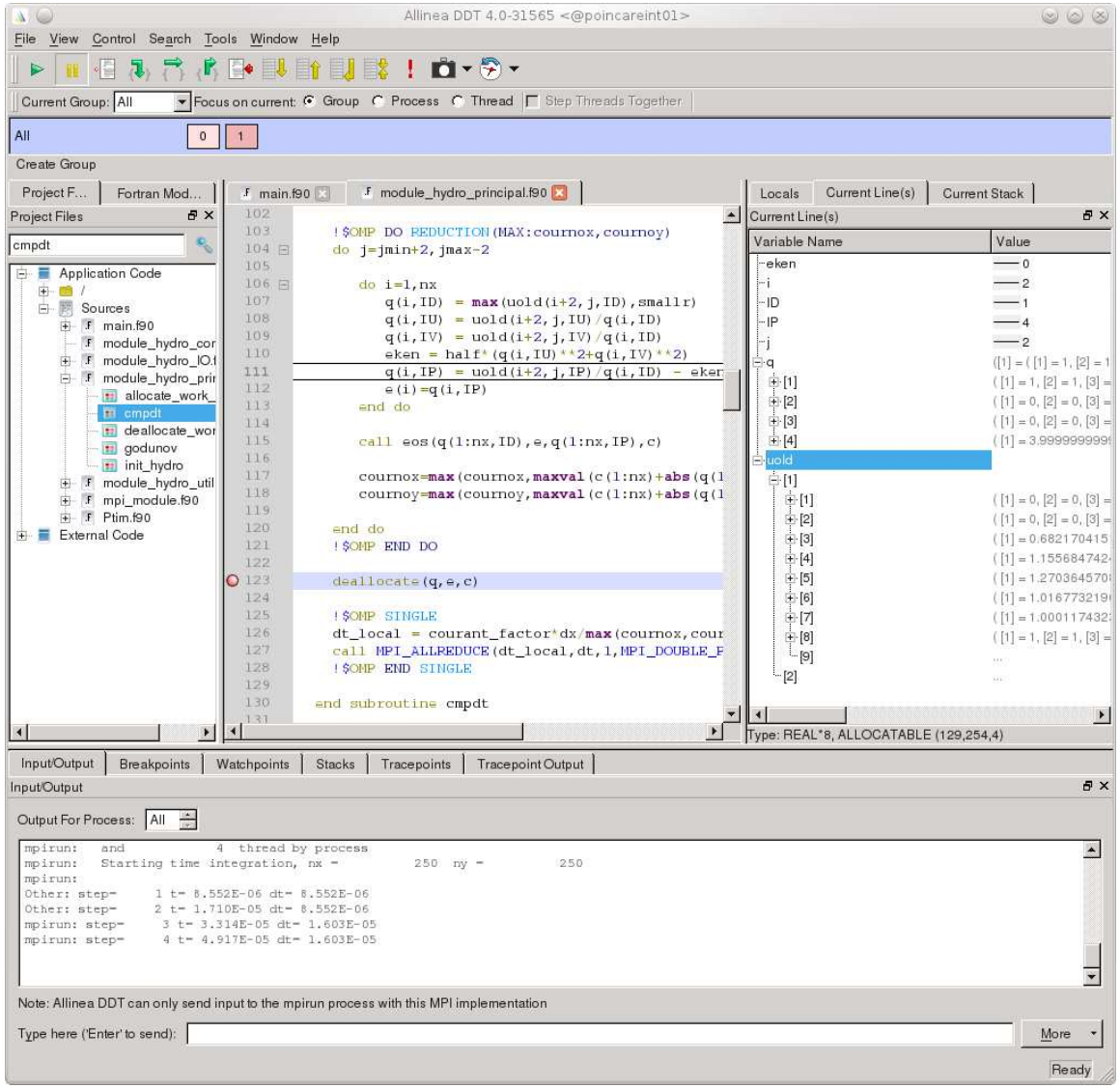


FIGURE 10 – Breakpoint for the visualisation of an array

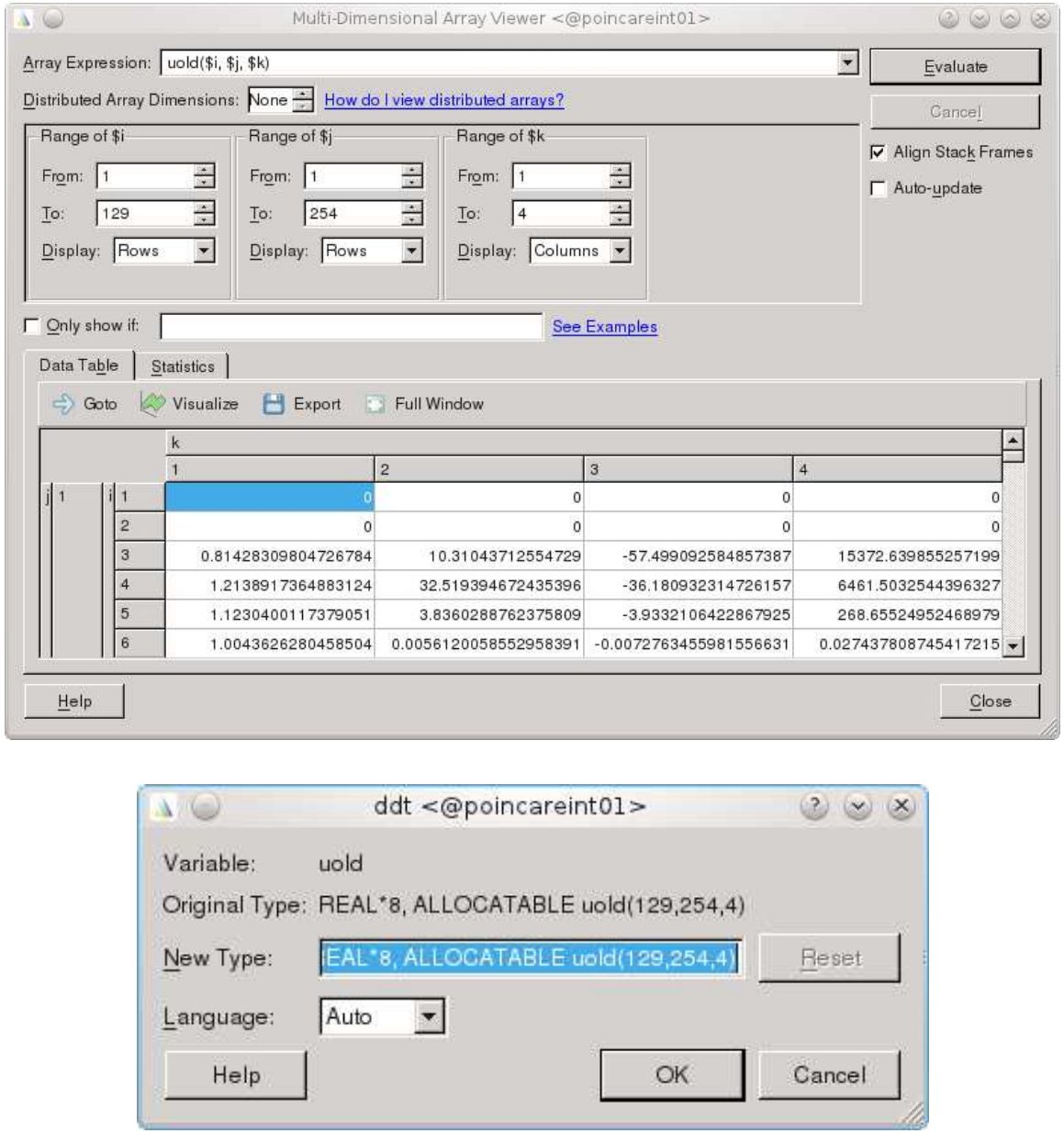


FIGURE 11 – Visualisation of an array

Conditional breakpoint

- Set the desired breakpoint.
- Right click on the breakpoint.
- Select *Edit breakpoint for All*.
- Select the program language (*C, Fortran*),
- In *Condition*, specify the break condition ; for example, in Fortran : *j.GE.100*.

Attachment on an execution code

- Submit a code execution in batch mode.
- Find the execution node :

```
Id                       Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
poincareint01-adm.26122- xxxxx        6/2  17:52 R  50  callmds      poincare060-adm
```

- Launch the debugger and click on *Attach to an already running program.*
- To add this node, click on *Choose Hosts.*

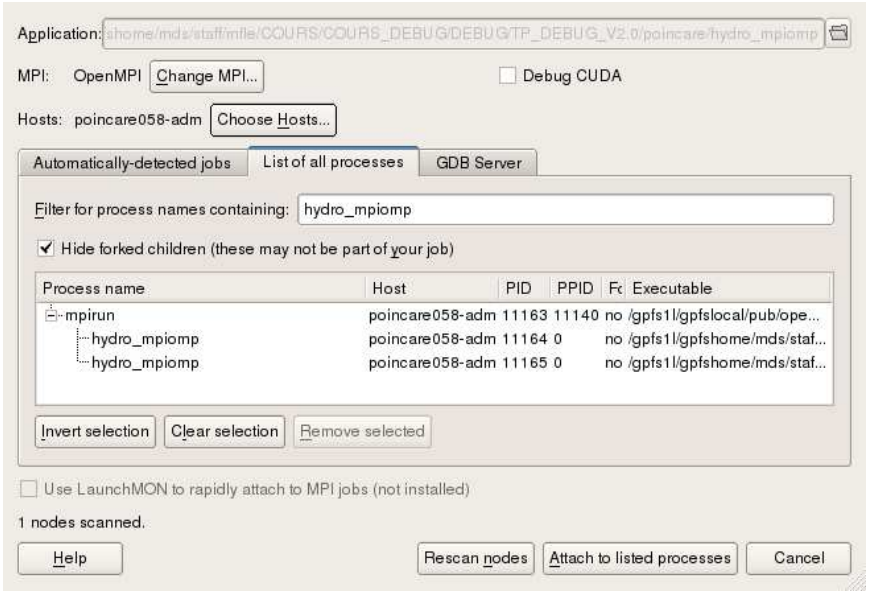


FIGURE 12 – Attachment of an application

Attachment on an execution code

- Click on *mpirun*, then on *Attach to selected processes.*

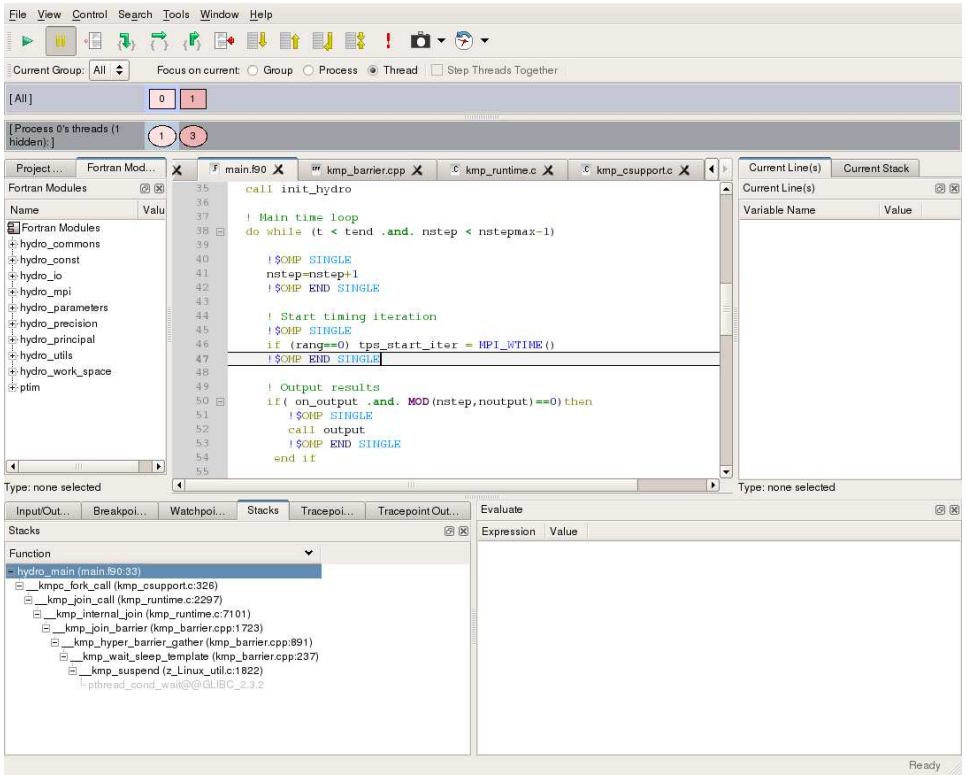


FIGURE 13 – Attachment of an application (continued)

TotalView allows debugging sequential and parallel codes :

- OpenMP
- MPI
- MPI + OpenMP or MPI + CUDA hybrid codes

Environment configuration

- *rm -rf ~/.totalview*
- *module load totalview*
- *export OMP_NUM_THREADS=nb_threads*
- *totalview &*

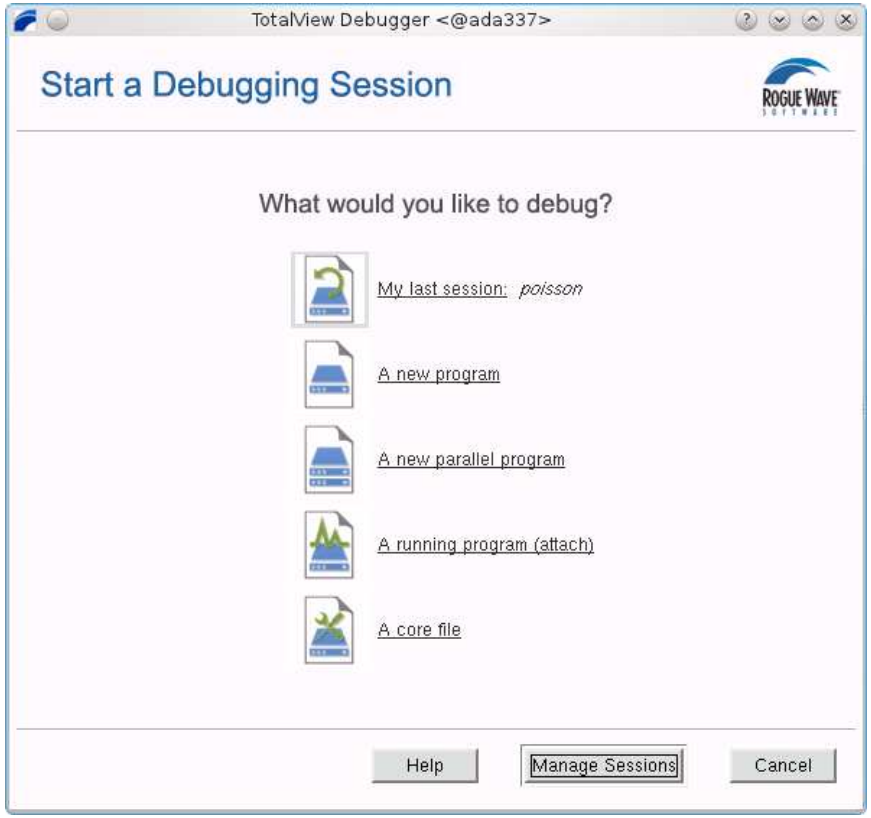


FIGURE 14 – Start window

- Click on *A new parallel program*.

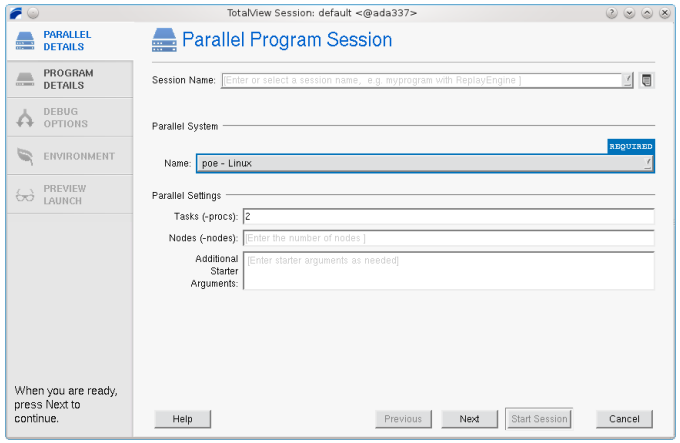


FIGURE 15 – Configuration of the execution

- Find "Parallel System", "Tasks".
- Click on *PROGRAM DETAILS*.

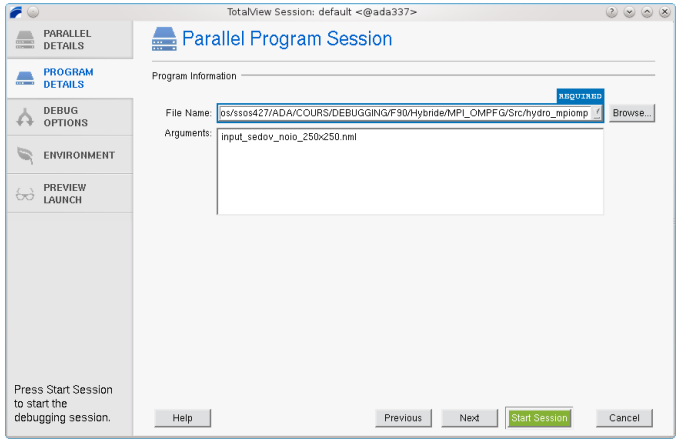


FIGURE 16 – Start of a parallel session

- Find "File Name" and "Arguments"
- Click on *DEBUG OPTIONS* and tick *Enable memory debugging*
- Click on *Start Session*

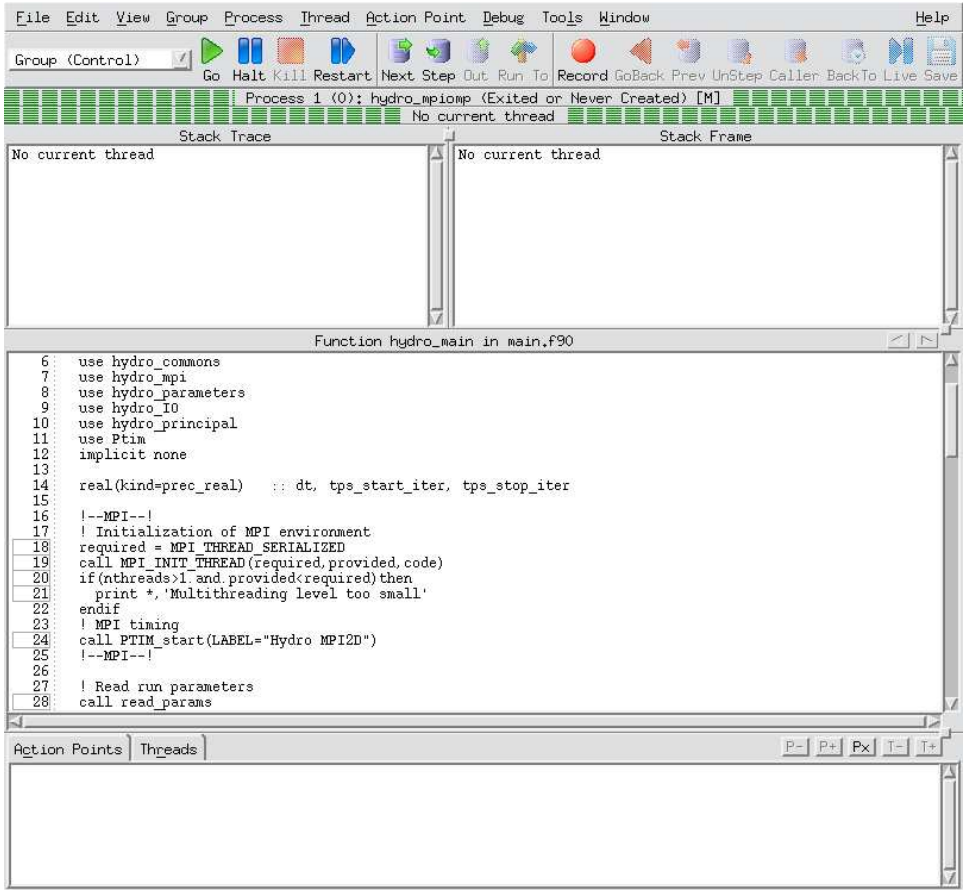


FIGURE 17 – Principal window

- Set a breakpoint and click on *Go*.
- The standard outputs of the program appear in the TotalView start window.

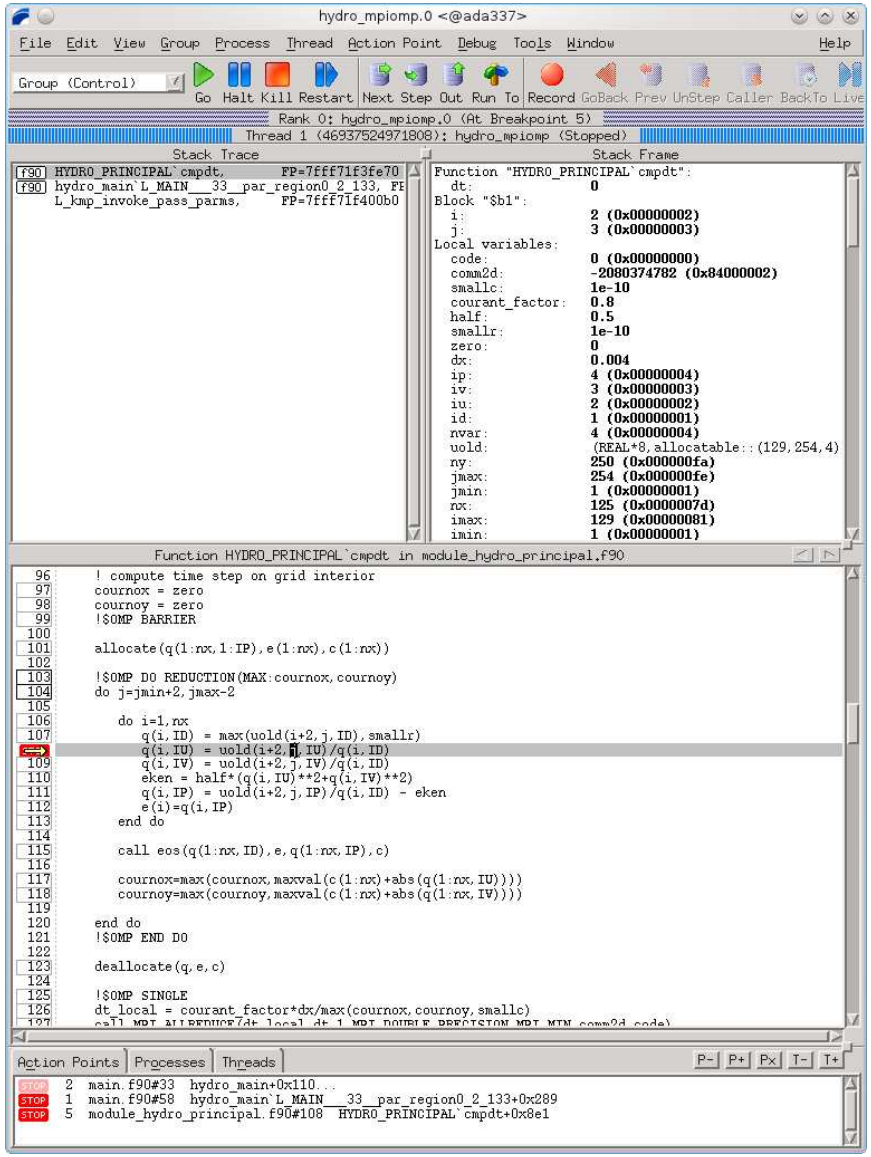


FIGURE 18 – Breakpoint in an OpenMP parallel region

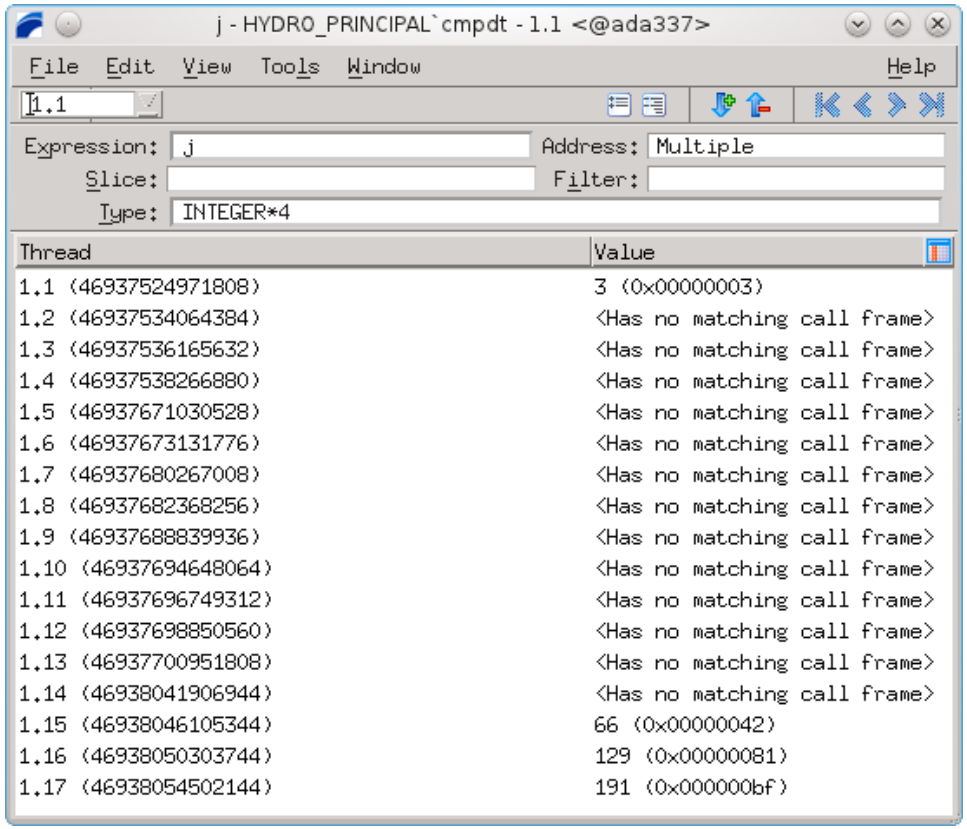


FIGURE 19 – Value of variable on each *thread*

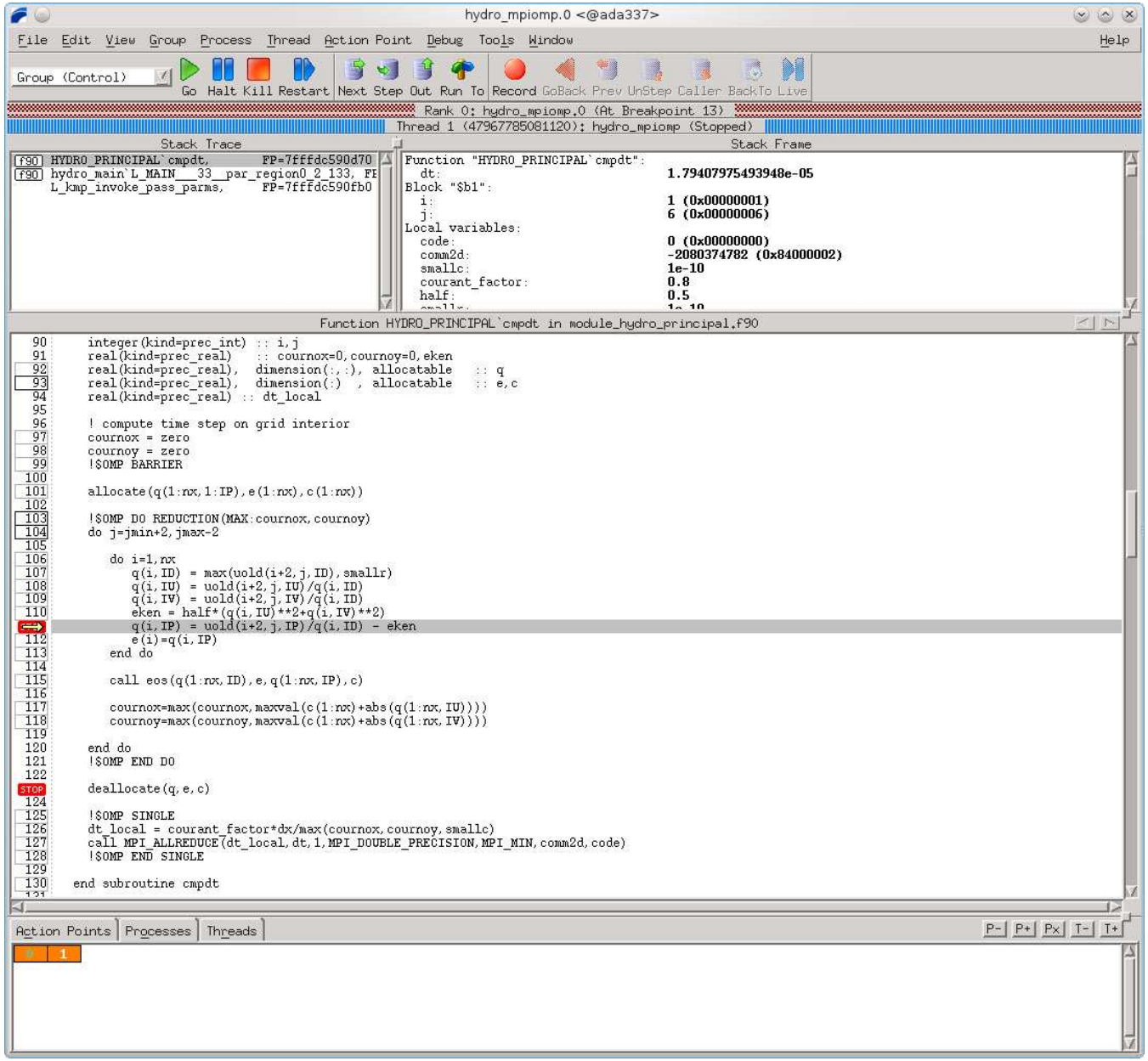
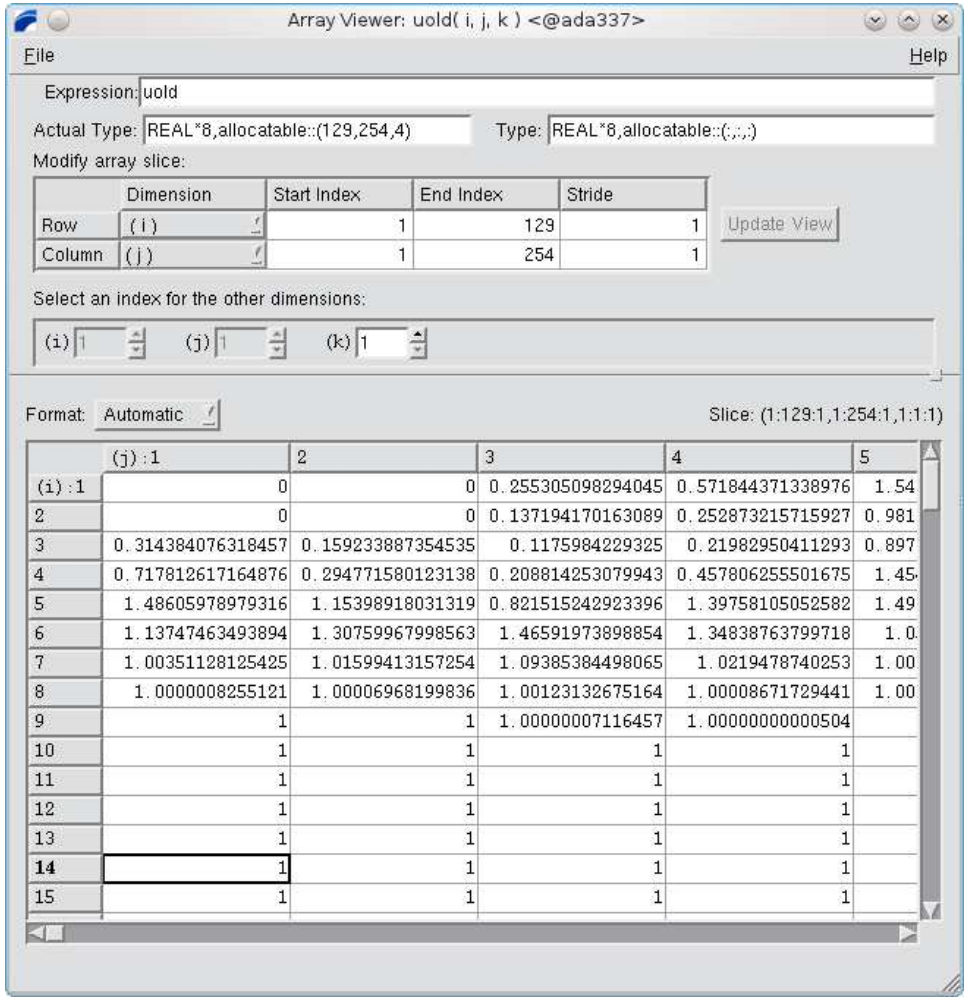


FIGURE 20 – Breakpoint for the visualisation of an array



	(j) :1	2	3	4	5
(i) :1	0	0	0.255305098294045	0.571844371338976	1.54
2	0	0	0.137194170163089	0.252873215715927	0.981
3	0.314384076318457	0.159233887354535	0.1175984229325	0.21982950411293	0.897
4	0.717812617164876	0.294771580123138	0.208814253079943	0.457806255501675	1.45
5	1.48605978979316	1.15398918031319	0.821515242923396	1.39758105052582	1.49
6	1.13747463493894	1.30759967998563	1.46591973898854	1.34838763799718	1.0
7	1.00351128125425	1.01599413157254	1.09385384498065	1.0219478740253	1.00
8	1.0000008255121	1.00006968199836	1.00123132675164	1.00008671729441	1.00
9	1	1	1.00000007116457	1.00000000000504	
10	1	1	1	1	
11	1	1	1	1	
12	1	1	1	1	
13	1	1	1	1	
14	1	1	1	1	
15	1	1	1	1	

FIGURE 21 – Visualisation of an array

Conditional breakpoint

- Set the desired breakpoint.
- Click on the breakpoint with the right button.
- Select *Properties*.
- Click on *Evaluate*,
- Select the program language (*C*, *Fortran*).
- In *Expression*, specify the condition of the break ; for example, in Fortran : *if (j > 100) $stop* or *$count 300*

Attachment on an execution code

- Submit the code execution in batch mode.
- Identify the execution node :

```
Id                      Owner      Submitted   ST PRI Class         Running On
----------------------- ---------- ----------- -- --- ------------ -----------
ada338.468484.0         xxxxxxx     9/12 16:57 R  100 c8t1         ada210-id
```

- Launch the debugger and click on *A running program(attach)*.
- Click on the *H+* button to add this node.

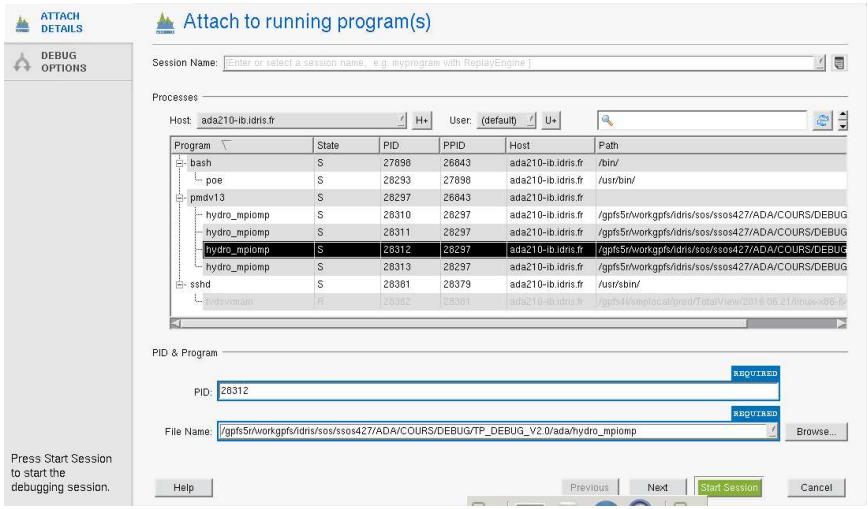


FIGURE 22 – Attachment of an application

Attachment on an execution code

- Click on *poe*, then on *Start Session*,

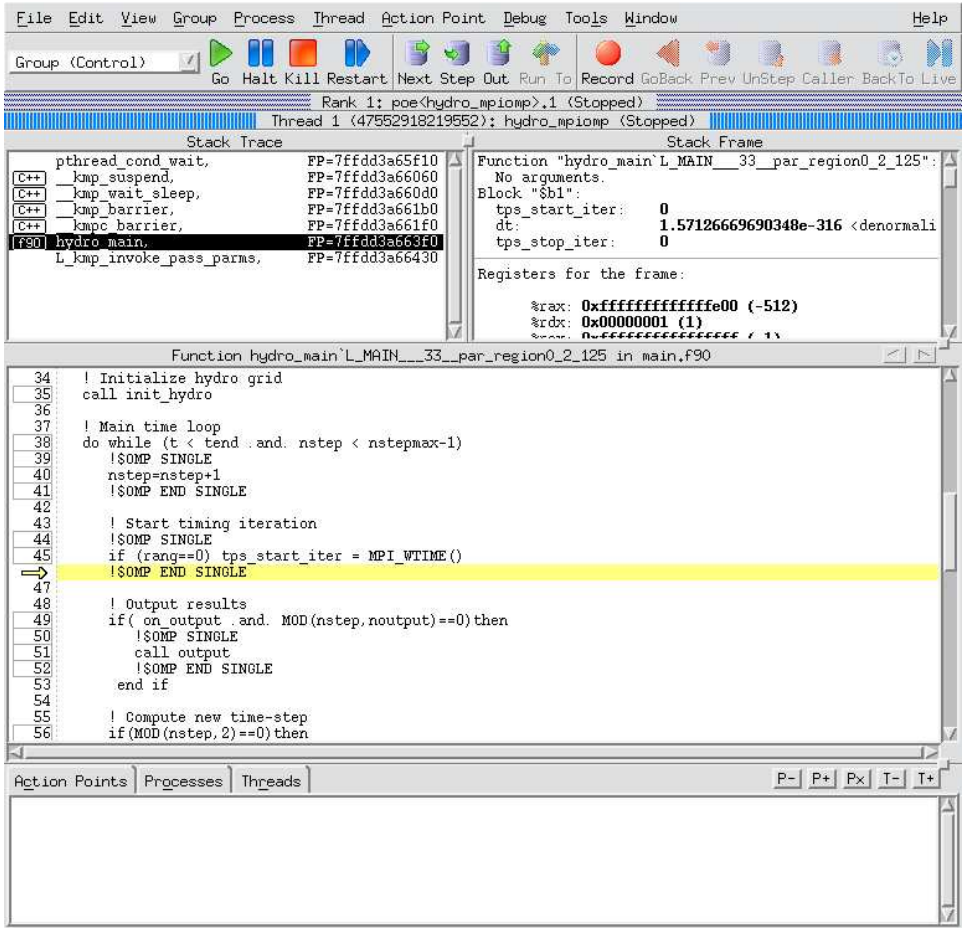


FIGURE 23 – Attachment of an application (cont'd)

5 – Hands-on exercises
5.1 – HYDRO code environment and implementation

- Platforms used :
  - For DDT : Cluster with 92 nodes having 16 processors each, and 4 GPU nodes (*poincare*, Maison de la Simulation).
  - For TotalView : Cluster of 332 nodes (*ada*, IDRIS).
- Load the environment : *. ./env.sh*
- Compilation : *make* (compilation for the debugging : *-O0 -g*)
- Execution in interactive : *./run.sh*
- Verification of the results in interactive : *./verif.sh nb_threads nb_procs*

5 – Hands-on exercises
5.2 – Exercice 1

The goal of the exercises is to obtain a correct version of the code with 4 MPI processes and 2 OpenMP threads.

- Test the code with 4 MPI processes and 2 OpenMP *threads*. What happens ?

5 – Hands-on exercises
5.2 – Exercice 1

Exercice 1 : Sequential debugging

- Execute the program with 1 MPI process, 1 OpenMP *thread*. What happens ?
- Use a debugger to locate this error.
- Correct the error.
- Verify that the sequential execution of the code (1 MPI process, 1 OpenMP *thread*) is correct.

5 – Hands-on exercises
5.3 – Exercice 2

Exercice 2 : Debugging in shared memory

- Test the OpenMP program with 1 MPI process, 4 OpenMP *threads*.
- What happens ?
- Execute the program via a debugger.
- When the program blocks (the results have ceased), stop the program and regard where the *threads* have stopped.
- Place a breakpoint on the line : *print *, "End of the loop"* , examine each *thread*.
- Correct the error.
- Verify that the program now runs until the end.

5 – Hands-on exercises
5.4 – Exercice 3

Exercice 3 : Debugging in shared memory

- Verify the results in OpenMP mode (1 MPI process, 16 OpenMP *threads*) with the *verif.sh* script.
- What happens ?
- Use a debugger to determine where the error is coming from.
- Find the variable from which *dt* is calulated.
- Find the location where it is calculated and used, and by which *threads*.
- Correct the error.
- Verify the results.

5 – Hands-on exercises
5.5 – Exercice 4

Exercice 4 : Debugging in distributed memory

- Verify the results in MPI mode (4 MPI processes, 1 OpenMP *thread*).
- What happens ?
- Use a debugger with 4 MPI processes, 1 OpenMP *thread*.
- Firstly, you can verify if each process has the correct neighbouring array values.
- Correct the error.
- Verify the results in MPI mode (4 MPI processes, 1 OpenMP *thread*).
- Now, verify the results with 4 MPI processes MPI and 2 OpenMP *threads*.

5 – Hands-on exercises
5.6 – Exercice 5

Exercice 5 : Debugging in distributed memory

- We are now going to test the program on a larger iteration number, 10 000 instead of the preceding 100.
- To do this, recopy the *input_sedov_noio_250x250_10000.nml* file input into *input_sedov_noio_250x250.nml*.
- Verify the results with this new input file with 4 MPI processes, 1 OpenMP *thread* via the *verif.sh* script.

```
./verif.sh 1 4 10000
```

- Some numeric differences appear around the 2000th iteration.
- **An error still remains ! ! ! !**

5 – Hands-on exercises
5.6 – Exercice 5

Exercice 5 : Debugging in distributed memory (cont'd)

- Use a debugger with 4 MPI processes, 1 OpenMP *thread*.
- Verify the communications : Do the received values correspond to the sent values ? To do this :
  - Run the program and click on *halt* shortly before the *step= 2000* display.
  - Create a conditional breakpoint to go to iteration 2000.
  - Then verify for each process that the correct data is exchanged between the processes.
- Correct the error.
- Now verify the results with 4 MPI processes, 1 OpenMP *thread*, then with 2 OpenMP *threads*.

Case of a code which does not function when there is a large number of execution *threads*

- This problem is often difficult to localise and even more so for hybrid codes.
- The parallel debuggers allow refining the zone on which you should concentrate.
- Example on HYDRO
  - Problem size : nx=2000, ny=4000
  - With 512 *threads* (256 MPI processes, 2 OpenMP *threads*), the code functions.
  - With 1024 *threads* (512 MPI processes, 2 OpenMP *threads* OpenMP), the code crashes.

```
+runjob --np 512 --envs OMP_NUM_THREADS=2 --ranks-per-node 16
        --exe ./hydro_mpiomp --args input_sedov_noio_2000x4000.nml

 MPI Execution with  512  processes ( 32 x 16 )
 and  2 threads by process
 Starting time integration, nx =  2000  ny =  4000
Global size of the domain nx*dims(1)*ny*dims(2)  -198967296
.....................................................................
```

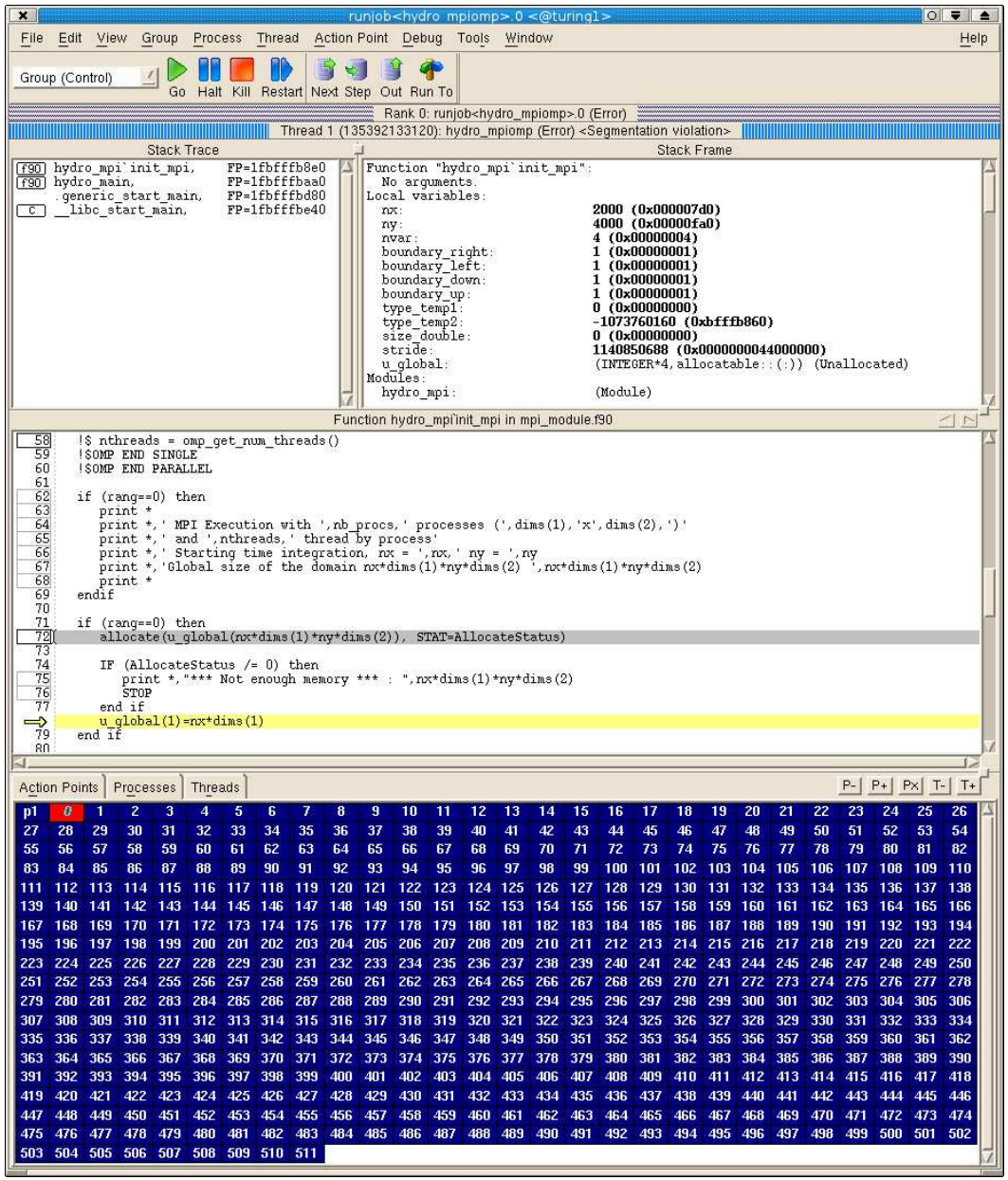


FIGURE 24 – Refine the zone where the problem is located.

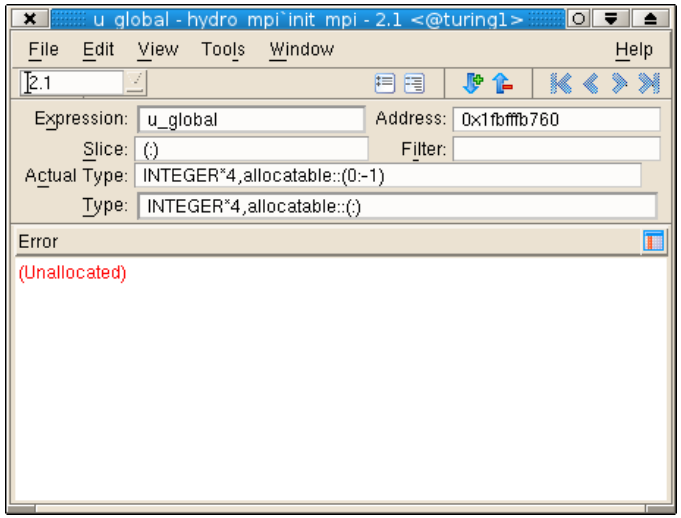


FIGURE 25 – u_global array

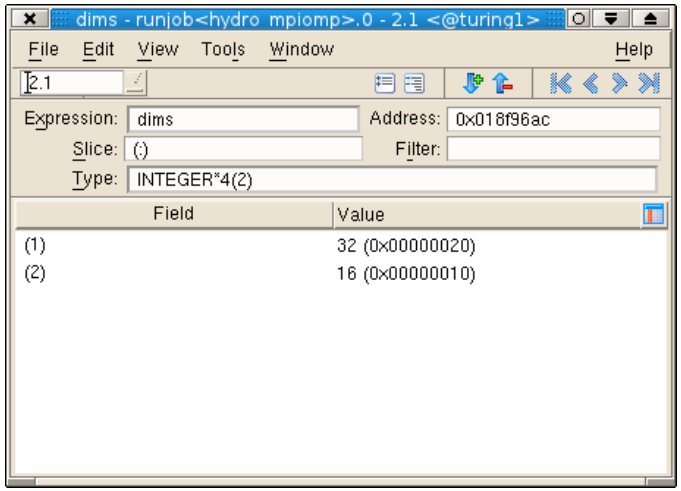


FIGURE 26 – Grid dimensions

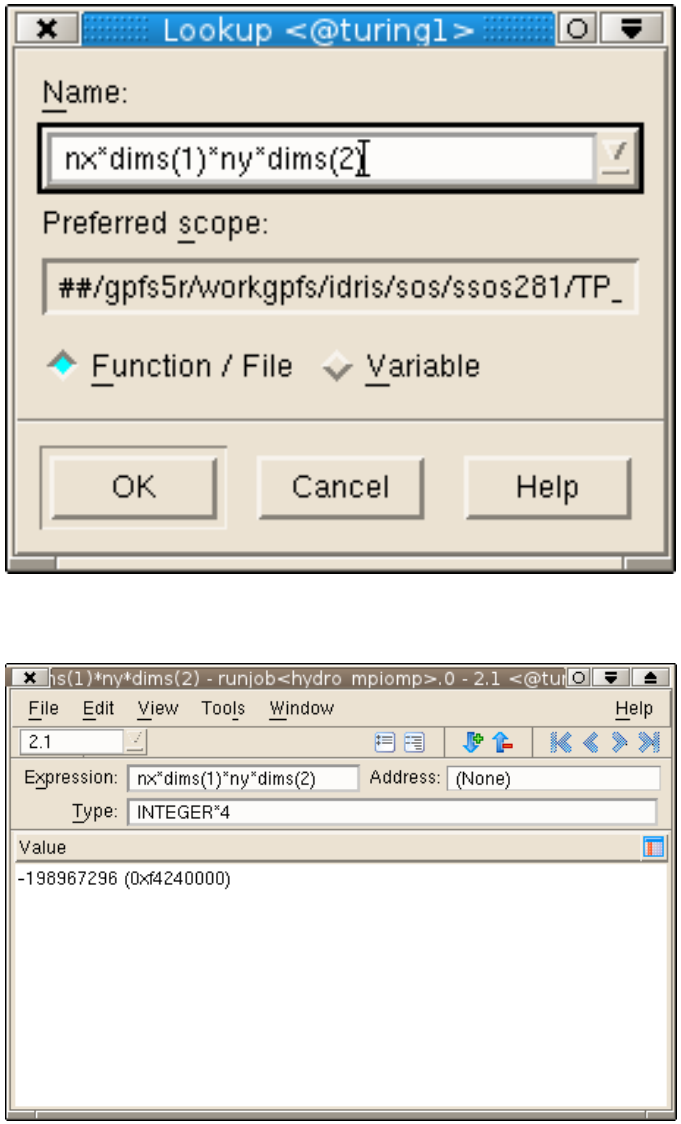


FIGURE 27 – A variable Lookup

- *"Debugging is twice as hard as writing the code in the first place".* Brian Kerningham (1974)
- Hence the advice :
  - Write the code well.
  - Write comments about it.
  - Test case.
  - Port it on several platforms.
  - ...
- **Attention : There is no miracle tool !**