

Parallel Debugger Utilisation

Isabelle DUPAYS
Marie FLÉ

Translated from French by Cynthia Taupin

Parallel Debugger Utilisation – Plan I

1	Introduction to parallel debugging.....	4
2	Some principles of parallel debugging.....	7
2.1	Shared memory model.....	8
2.2	Distributed memory model.....	14
2.3	Hybrid model.....	19
2.4	Methodology.....	21
3	Presentation of the HYDRO code.....	22
3.1	Description.....	23
3.2	Calculation of the solution.....	24
3.3	Hybrid version of the HYDRO code.....	25
4	HPC debugging tools.....	27
4.1	Introduction.....	28
4.2	DDT.....	29
4.3	TotalView.....	41
5	Hands-on exercises.....	54
5.1	HYDRO code environment and implementation.....	55
5.2	Exercice 1.....	56
5.3	Exercice 2.....	58
5.4	Exercice 3.....	59
5.5	Exercice 4.....	60
5.6	Exercice 5.....	61
6	Large scale debugging.....	63

7 Conclusion	68
--------------------	----

- 1 Introduction to parallel debugging
- 2 Some principles of parallel debugging
- 3 Presentation of the HYDRO code
- 4 HPC debugging tools
- 5 Hands-on exercises
- 6 Large scale debugging
- 7 Conclusion

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.

1	Introduction to parallel debugging	
2	Some principles of parallel debugging	
2.1	Shared memory model	8
2.2	Distributed memory model	14
2.3	Hybrid model.....	19
2.4	Methodology	21
3	Presentation of the HYDRO code	
4	HPC debugging tools	
5	Hands-on exercises	
6	Large scale debugging	
7	Conclusion	

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=40000000
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
fortrtl: 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

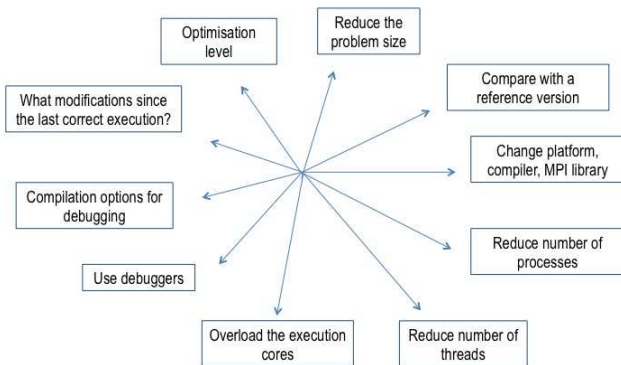


FIGURE 1 – Debugging diagram

1	Introduction to parallel debugging	
2	Some principles of parallel debugging	
3	Presentation of the HYDRO code	
3.1	Description	23
3.2	Calculation of the solution	24
3.3	Hybrid version of the HYDRO code	25
4	HPC debugging tools	
5	Hands-on exercises	
6	Large scale debugging	
7	Conclusion	

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

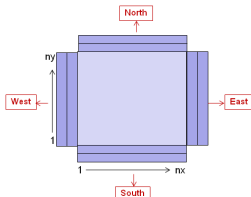
3.2 – Calculation of the solution

- 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

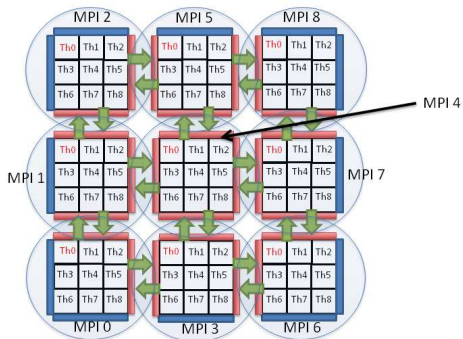


FIGURE 3 – Hybrid version of the HYDRO code

1	Introduction to parallel debugging	
2	Some principles of parallel debugging	
3	Presentation of the HYDRO code	
4	HPC debugging tools	
4.1	Introduction	28
4.2	DDT	29
4.3	TotalView	41
5	Hands-on exercises	
6	Large scale debugging	
7	Conclusion	

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*

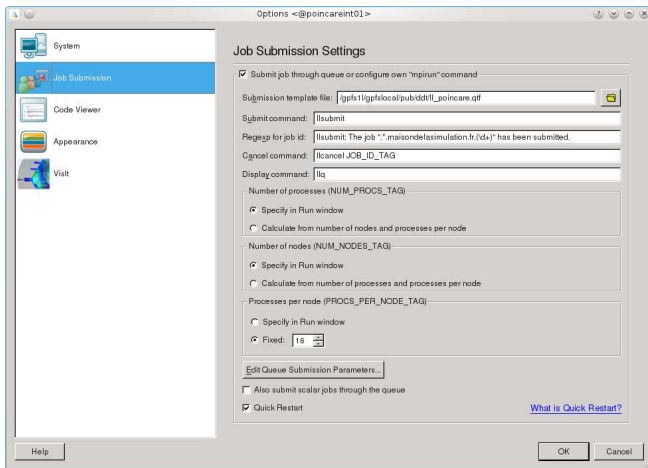


FIGURE 5 – Configuration for job submission

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

Application: igpfshome/mds/grprtraining/training30/TP_DEBUG_PARALL/poincare_ Details

Application: File icon

Arguments: Dropdown

stdin file: File icon

Working Directory: File icon

MPI: 2 processes, 1 node, OpenMPI Details

Number of processes: Dropdown Number of Nodes: Dropdown

Implementation: OpenMPI, use queue Change...

mpirun arguments Dropdown

OpenMP: 4 threads Details

Number of OpenMP threads: Dropdown

CUDA Details

Memory Debugging Details...

Queue Submission Parameters: Job Type=mpich, Wall Clock Limit=00:30:00, Nod Details...

Environment Variables: none Details

Plugins: none Details

Help Submit Cancel

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*

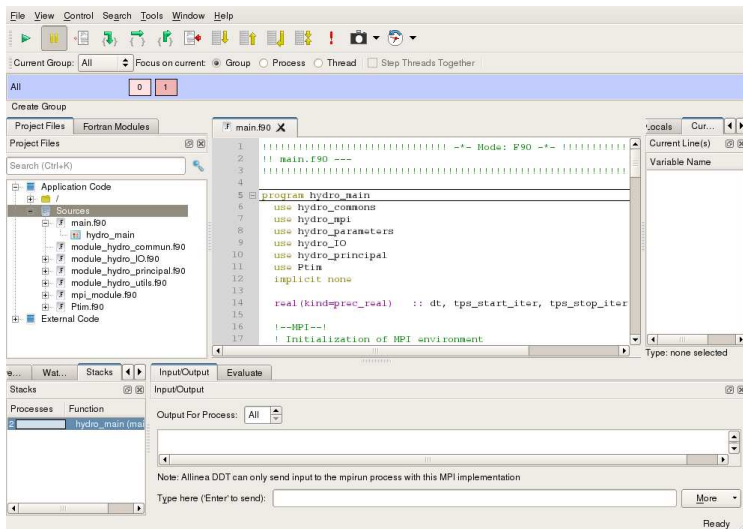


FIGURE 7 – Principal window

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

The screenshot shows the Alinea DDT debugger interface. The main window displays the Fortran source code for `main.f90` and `module_hydro_principal.f90`. A breakpoint is set on line 108, which is inside an OpenMP parallel region. The code snippet is as follows:

```

98  cournoy = zero
99  !$OMP BARRIER
100
101  allocate (q(1:nx, 1:IP), e(1:nx), c(1:nx))
102
103  !$OMP DO REDUCTION(MAX: cournox, cournoy)
104  do j=jmin+2, jmax-2
105
106  do i=1,nx
107  q(i, ID) = max(uold(i+2, j, ID), smallr)
108  q(i, IU) = uold(i+2, j, IU) / q(i, ID)
109  q(i, IV) = uold(i+2, j, IV) / q(i, ID)
110  eken = half * (q(i, IU)**2 + q(i, IV)**2)
111  q(i, IP) = uold(i+2, j, IP) / q(i, ID) - eken
112  e(i) = q(i, IP)
113  end do
114
115  call eos(q(1:nx, ID), e, q(1:nx, IP), c)
116
117  cournox = max(cournox, maxval(c(1:nx) + abs(q(1:nx, IU))))
118  cournoy = max(cournoy, maxval(c(1:nx) + abs(q(1:nx, IV))))

```

The left sidebar shows the project structure, including the source files and modules. The bottom panel shows the output for the current process, which includes the following information:

```

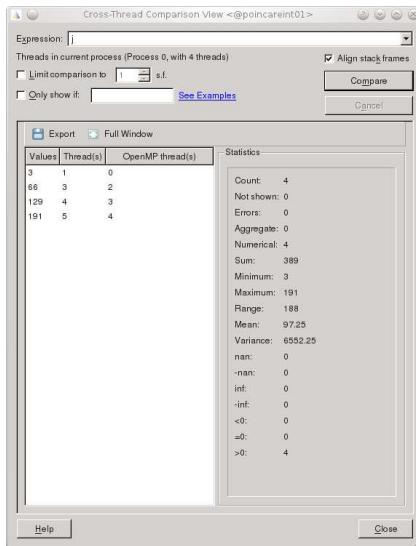
Output For Process: All
-----
mpirun: MPI Execution with 2 processes ( 2 x 1 )
mpirun: and 4 thread by process
mpirun: Starting time integration, nx = 250 ny = 250
mpirun:
Other: step- 1 t= 9.881-324 dt= 9.881-324
Other: step- 2 t= 1.976-323 dt= 9.881-324

```

Note: Allinea DDT can only send input to the mpirun process with this MPI implementation

Type here (Enter to send): More Ready

FIGURE 8 – Breakpoint in an OpenMP parallel region

FIGURE 9 – Value of variable on each *thread*

The screenshot shows the Alinea DDT 4.0-31565 debugger interface. The main window displays Fortran code with a red circle highlighting a breakpoint set on the `deallocate(q, e, c)` statement at line 123. The right-hand pane shows the current state of memory, with a highlighted row indicating the address of the deallocated array.

Current Line(s)

Variable Name	Value
-kern	0
-i	2
-IP	4
-i	2
-q	([1] = [1] - 1, [2] = [1] - 1, [3] = [1] - 1, [4] = [1] - 1, [5] = [1] - 1)
+ [0]	([1] = 0, [2] = 0, [3] = 0, [4] = 0, [5] = 0)
+ [1]	([1] = 0, [2] = 0, [3] = 0, [4] = 0, [5] = 0)
+ [2]	([1] = 0, [2] = 0, [3] = 0, [4] = 0, [5] = 0)
+ [3]	([1] = 0, [2] = 0, [3] = 0, [4] = 0, [5] = 0)
+ [4]	([1] = 3.999999999999999)
-old	([1] = 1.270364570)
+ [1]	([1] = 0, [2] = 0, [3] = 0, [4] = 0, [5] = 0)
+ [2]	([1] = 0, [2] = 0, [3] = 0, [4] = 0, [5] = 0)
+ [3]	([1] = 0.682176415)
+ [4]	([1] = 1.55684742)
+ [5]	([1] = 1.270364570)
+ [6]	([1] = 1.016773219)
+ [7]	([1] = 1.000117432)
+ [8]	([1] = 1, [2] = 1, [3] = 1, [4] = 1, [5] = 1)
+ [9]	...
+ [10]	...

Type: REAL*8, ALLOCATABLE (128,254,4)

Output For Process: All

```

mpirun: and 4 thread by process
mpirun: Starting time integration, n = 250 ny = 250
mpirun:
Other: step= 1 t= 8.552E-04 dt= 8.552E-04
Other: step= 2 t= 1.710E-03 dt= 8.552E-04
mpirun: step= 3 t= 3.314E-03 dt= 1.603E-03
mpirun: step= 4 t= 8.937E-03 dt= 3.603E-03
  
```

Now: Alinea DDT can only send input to the mpirun process with this MPI implementation

Type here (Enter to send): More Ready

FIGURE 10 – Breakpoint for the visualisation of an array

Multi-Dimensional Array Viewer <@poincareint01>

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

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

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

Range of \$j: From: 1 To: 254 Display: Rows

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

Align Stack Frames
 Auto-update

Only show if: See Examples

Data Table | Statistics

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

		k			
		1	2	3	4
i	1	0	0	0	0
	2	0	0	0	0
	3	0.81428309804726784	10.31048712554729	-57.499092584857387	15372.639855257199
	4	1.2138917364883124	32.519394672435396	-36.180932314726157	6461.5032544396327
	5	1.1230400117379051	3.8360288762375809	-3.9332106422867925	266.65524952468979
	6	1.0043626280458504	0.0056120058552958391	-0.0072763455981556631	0.027437868745417215

Help Close

ddt <@poincareint01>

Variable: `uold`

Original Type: `REAL*8, ALLOCATABLE uold(129,254,4)`

New Type: `REAL*8, ALLOCATABLE uold(129,254,4)` Reset

Language: `Auto`

Help OK Cancel

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	clallm	poincare060-adm

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

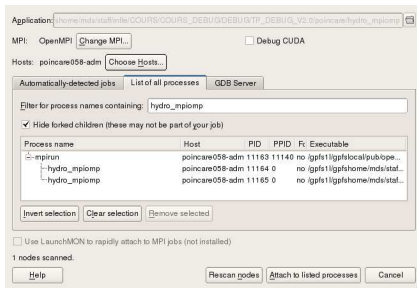


FIGURE 12 – Attachment of an application

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 &*

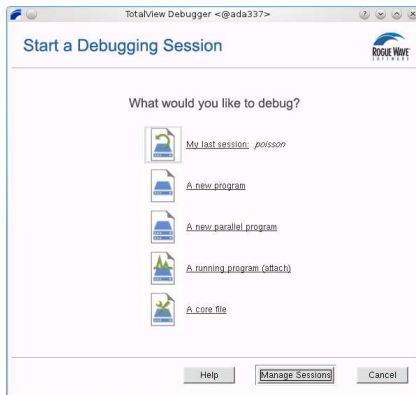


FIGURE 14 – Start window

- Click on *A new parallel program*.

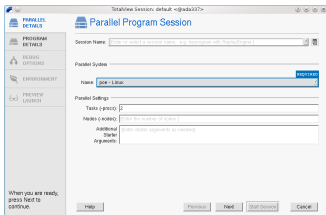


FIGURE 15 – Configuration of the execution

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

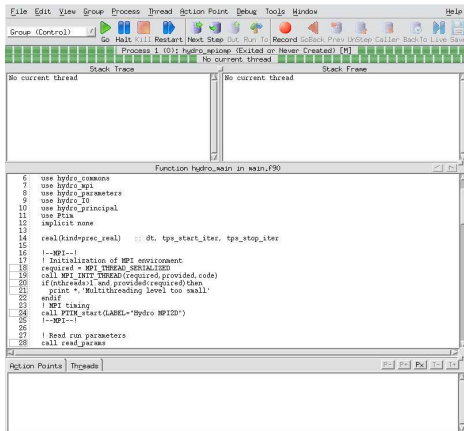


FIGURE 17 – Principal window

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

The screenshot shows the TotalView debugger interface for the application 'hydro_mpiomp.0'. The main window displays the source code for 'HYDRO_PRINCIPAL.c' with a breakpoint set at line 109. The code is as follows:

```

96 | compute time step on grid interior
97 | cournum = zero
98 | courney = zero
99 | $OMP BANKER
100 |
101 | allocate(q(1:nx,1:ny),e(1:nx),c(1:nx))
102 |
103 | $OMP DO REDUCTION(MAX:cournum,courney)
104 | do j=jmin+2,jmax-2
105 |
106 |   do i=1,nx
107 |     q(i,1D) = max(uold(i+2,1D),small)
108 |     q(i,2U) = uold(i+2,2U)/q(i,1D)
109 |     q(i,1V) = uold(i+2,1V)/q(i,1D)
110 |     skm = half*(q(i,1D)+c*(1,1V)+1)
111 |     q(i,1P) = uold(i+2,1P)/q(i,1D) = skm
112 |     s(i)=q(i,1P)
113 |   end do
114 |
115 |   call esp(q(1:nx,1D),e,q(1:nx,1P),c)
116 |   cournum=max(cournum,maxval(c(1:nx),abs(q(1:nx,1D))))
117 |   courney=max(courney,maxval(c(1:nx),abs(q(1:nx,1V))))
118 |
119 | end do
120 | $OMP END DO
121 |
122 | deallocate(q,e,c)
123 |
124 | $OMP SIMPLX
125 | dt_local = courant_factor*dt/max(cournum,courney,small)
126 | --(1) SET AT ENTRY OF LOCAL DO 1 SET SOURCE DESTINUM SET WITH COMPILED CODE
127 |

```

The 'Stack Trace' panel shows the current function 'HYDRO_PRINCIPAL.c' and its caller 'HYDRO_PRINCIPAL.c'.

The 'Stack Frame' panel shows the following variables:

```

Block "tbl":
dt: 0
i: 1
j: 3 (0x00000003)
Local variables:
code: 0 (0x00000000)
com2d: -2580374782 (0x40000002)
e: 1e-10
courant_factor: 0.8
half: 0.5
small: 1e-10
zero: 0
dt: 0.004
ip: 4 (0x00000004)
iv: 3 (0x00000003)
jv: 2 (0x00000002)
id: 1 (0x00000001)
nx: 4 (0x00000004)
ny: 250 (0x000001fa)
jmax: 254 (0x000001fe)
jmin: 1 (0x00000001)
nx: 125 (0x0000007d)
ny: 123 (0x0000007b)
jmin: 1 (0x00000001)

```

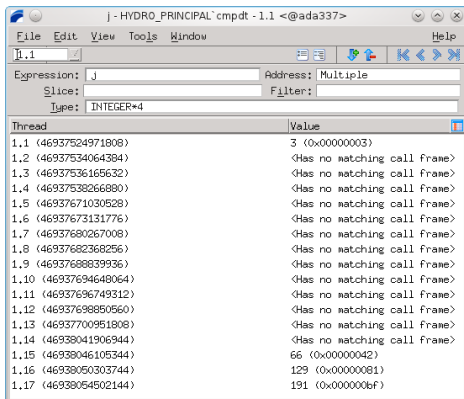
The 'Action Points' panel shows the following processes and threads:

```

2 | main [90#13] hydro_main+0x110
1 | main [90#58] hydro_main.L.MAIN_33_par_region0_2_133+0x289
5 | module_hydro_principal_f90#108 HYDRO_PRINCIPAL.c'pdt+0x8e1

```

FIGURE 18 – Breakpoint in an OpenMP parallel region



The screenshot shows the TotalView debugger interface for a process named 'j - HYDRO_PRINCIPAL`cmpdt -1.1 <@ada337>'. The 'Expression' field contains 'J', the 'Address' is 'Multiple', and the 'Type' is 'INTEGER*4'. Below this, a table displays the values of variable 'J' for 17 different threads. Most threads have a value of 3, while threads 1.15, 1.16, and 1.17 have values of 66, 129, and 191 respectively. Threads 1.2 through 1.14 show '<Has no matching call frame>'. The table has columns for 'Thread' and 'Value'.

Thread	Value
1.1 (46937524971808)	3 (0x00000003)
1.2 (46937534064384)	<Has no matching call frame>
1.3 (46937536165632)	<Has no matching call frame>
1.4 (46937538266880)	<Has no matching call frame>
1.5 (46937671030528)	<Has no matching call frame>
1.6 (46937673131776)	<Has no matching call frame>
1.7 (46937680267008)	<Has no matching call frame>
1.8 (46937682368256)	<Has no matching call frame>
1.9 (4693768839936)	<Has no matching call frame>
1.10 (46937694648064)	<Has no matching call frame>
1.11 (46937696749312)	<Has no matching call frame>
1.12 (46937698850560)	<Has no matching call frame>
1.13 (46937700951808)	<Has no matching call frame>
1.14 (46938041906944)	<Has no matching call frame>
1.15 (46938046105344)	66 (0x00000042)
1.16 (46938050303744)	129 (0x00000081)
1.17 (46938054502144)	191 (0x000000bf)

FIGURE 19 – Value of variable on each *thread*

The screenshot shows the TotalView parallel debugger interface. The main window displays the source code for the 'HYDRO_PRINCIPAL' module, with a red highlight on line 110: `e(i) = half*(q(i,iu)**2+q(i,iv)**2)`. The 'Stack Trace' and 'Stack Frame' panels show the current function call and its arguments. The 'Action Points' panel at the bottom shows a single breakpoint set at the highlighted line.

Stack Trace:

Function	Address
Function "HYDRO_PRINCIPAL" cmptdt	1.79407975493940e-05

Stack Frame:

Variable	Value
dt	1.79407975493940e-05
Block "\$b1"	
i	1 (0x00000001)
j	6 (0x00000006)
Local variables:	
code	0 (0x00000000)
count2d	-2080374782 (0x84000002)
smallc	1e-10
courant_factor	0.0
half	0.5
oc31a	1e-10

Function HYDRO_PRINCIPAL'cmptdt in module HYDRO_principal.F90:

```

90 integer(kind=prec_int) :: i,j
91 real(kind=prec_real) :: courmax=0,cournoy=0,eken
92 real(kind=prec_real), dimension(:, :), allocatable :: q
93 real(kind=prec_real), dimension(:, :), allocatable :: e,c
94 real(kind=prec_real) :: dt_local
95
96 ! compute time step on grid interior
97 courmax = zero
98 cournoy = zero
99 !$OMP BARRIER
100
101 allocate(q(1.nx,1.iy),e(1.nx),c(1.nx))
102
103 !$OMP DO REDUCTION(OMAX: courmax, cournoy)
104 do j=jmin=2,jmax=2
105
106     do i=1,nx
107         q(i,iu) = max(vold(i+2,j,iu),smallc)
108         q(i,iv) = vold(i+2,j,iv)/q(i,iu)
109         eken = half*(q(i,iu)**2+q(i,iv)**2)
110         e(i,iu) = vold(i+2,j,iu)/q(i,iu) - eken
111     end do
112
113     call eos(q(1.nx,iu),e,q(1.nx,iv),c)
114
115     courmax=max(courmax,maxval(c(1.nx)+abs(q(1.nx,iu))))
116     cournoy=max(cournoy,maxval(c(1.nx)+abs(q(1.nx,iv))))
117
118 end do
119 !$OMP END DO
120
121 deallocate(q,e,c)
122
123 !$OMP SINGLE
124 dt_local = courant_factor*dt/max(courmax,cournoy,smallc)
125 call MPI_ALLREDUCE(dt_local,dt,1,MPI_DOUBLE_PRECISION,MPI_MIN,comm2d,code)
126 !$OMP END SINGLE
127
128 end subroutine cmptdt

```

Action Points:

Line	Condition
110	

FIGURE 20 – Breakpoint for the visualisation of an array

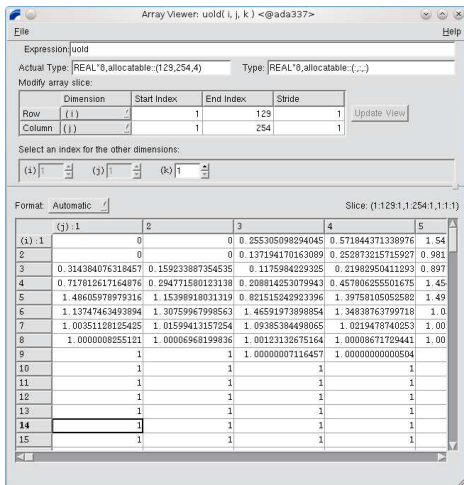


FIGURE 21 – Visualisation of an array

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

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

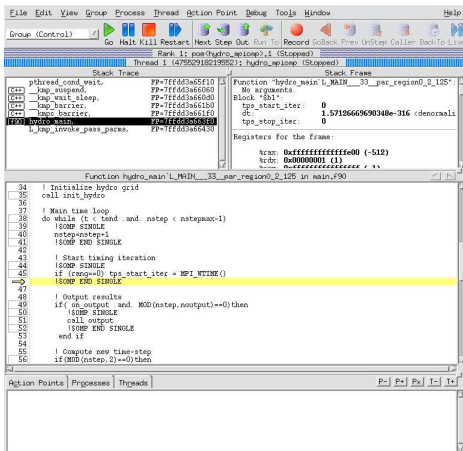


FIGURE 23 – Attachment of an application (cont'd)

1	Introduction to parallel debugging	
2	Some principles of parallel debugging	
3	Presentation of the HYDRO code	
4	HPC debugging tools	
5	Hands-on exercises	
5.1	HYDRO code environment and implementation	55
5.2	Exercice 1	56
5.3	Exercice 2	58
5.4	Exercice 3	59
5.5	Exercice 4	60
5.6	Exercice 5	61
6	Large scale debugging	
7	Conclusion	

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 calculated.
- 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 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*.

- 1 Introduction to parallel debugging
- 2 Some principles of parallel debugging
- 3 Presentation of the HYDRO code
- 4 HPC debugging tools
- 5 Hands-on exercises
- 6 Large scale debugging**
- 7 Conclusion

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
.....
```


Rank 0: tunjob=hydro_mpiomp>0 (Error)
Thread 1 (135392133120): hydro_mpiomp (Error) <Segmentation violation>

Stack Trace

PC	Address	Function
000	hydro_mpi_init_mpi	Function "hydro_mpi_init_mpi"
000	hydro_main	No arguments.
	generic_start_main	Local variables:
	_libc_start_main	ncx: 2000 (0x00007f00)
		ny: 4000 (0x00007fa0)
		ncz: 4 (0x00000004)
		boundary_right: 1 (0x00000001)
		boundary_left: 1 (0x00000001)
		boundary_down: 1 (0x00000001)
		boundary_up: 1 (0x00000001)
		type_trap1: 0 (0x00000000)
		type_trap2: -107768160 (0x0ffff860)
		size_double: 0 (0x00000000)
		stride: 114052608 (0x0000000040000000)
		v_global: (13780284, allocatable (:)) (Unallocated)
		Modules:
		hydro_mpi: (Module)

Function hydro_mpi_init_mpi in mpi_module.f90

```

50  !$ nthreads = emp_get_num_threads()
51  !$OMP END STACK
52  !$OMP END PARALLEL
53
54  if (rank==0) then
55    print *
56    print *, 'MPI Execution with ', nb_procs, ' processes (' ,dias(1), 'x', dias(2), ') '
57    print *, ' and ', nthreads, ' thread by process'
58    print *, ' Starting time integration, ncx = ', ncx, ' ny = ', ny
59    print *, ' Global size of the domain ncz*dias(1)*ny*dias(2) , ncz*dias(1)*ny*dias(2)
60    print *
61  endif
62
63  if (rank==0) then
64    allocate(v_global(ncz*dias(1)*ny*dias(2)), STAT=allocateStatus)
65    if (AllocateStatus /= 0) then
66      print *, '*** Not enough memory *** ', ncx*dias(1)*ny*dias(2)
67      STOP
68    endif
69    v_global(1)*ncx*dias(1)
70  end if
71
72  do
73  end do

```

Action Points	Processes	Threads
p1	0	1
2	2	3
3	4	5
4	6	7
5	8	9
6	10	11
7	12	13
8	14	15
9	16	17
10	18	19
11	20	21
12	22	23
13	24	25
14	26	27
15	28	29
16	30	31
17	32	33
18	34	35
19	36	37
20	38	39
21	40	41
22	42	43
23	44	45
24	46	47
25	48	49
26	50	51
27	52	53
28	54	55
29	56	57
30	58	59
31	60	61
32	62	63
33	64	65
34	66	67
35	68	69
36	70	71
37	72	73
38	74	75
39	76	77
40	78	79
41	80	81
42	82	83
43	84	85
44	86	87
45	88	89
46	90	91
47	92	93
48	94	95
49	96	97
50	98	99
51	100	101
52	102	103
53	104	105
54	106	107
55	108	109
56	110	111
57	112	113
58	114	115
59	116	117
60	118	119
61	120	121
62	122	123
63	124	125
64	126	127
65	128	129
66	130	131
67	132	133
68	134	135
69	136	137
70	138	139
71	140	141
72	142	143
73	144	145
74	146	147
75	148	149
76	150	151
77	152	153
78	154	155
79	156	157
80	158	159
81	160	161
82	162	163
83	164	165
84	166	167
85	168	169
86	170	171
87	172	173
88	174	175
89	176	177
90	178	179
91	180	181
92	182	183
93	184	185
94	186	187
95	188	189
96	190	191
97	192	193
98	194	195
99	196	197
100	198	199
101	200	201
102	202	203
103	204	205
104	206	207
105	208	209
106	210	211
107	212	213
108	214	215
109	216	217
110	218	219
111	220	221
112	222	223
113	224	225
114	226	227
115	228	229
116	230	231
117	232	233
118	234	235
119	236	237
120	238	239
121	240	241
122	242	243
123	244	245
124	246	247
125	248	249
126	250	251
127	252	253
128	254	255
129	256	257
130	258	259
131	260	261
132	262	263
133	264	265
134	266	267
135	268	269
136	270	271
137	272	273
138	274	275
139	276	277
140	278	279
141	280	281
142	282	283
143	284	285
144	286	287
145	288	289
146	290	291
147	292	293
148	294	295
149	296	297
150	298	299
151	300	301
152	302	303
153	304	305
154	306	307
155	308	309
156	310	311
157	312	313
158	314	315
159	316	317
160	318	319
161	320	321
162	322	323
163	324	325
164	326	327
165	328	329
166	330	331
167	332	333
168	334	335
169	336	337
170	338	339
171	340	341
172	342	343
173	344	345
174	346	347
175	348	349
176	350	351
177	352	353
178	354	355
179	356	357
180	358	359
181	360	361
182	362	363
183	364	365
184	366	367
185	368	369
186	370	371
187	372	373
188	374	375
189	376	377
190	378	379
191	380	381
192	382	383
193	384	385
194	386	387
195	388	389
196	390	391
197	392	393
198	394	395
199	396	397
200	398	399
201	400	401
202	402	403
203	404	405
204	406	407
205	408	409
206	410	411
207	412	413
208	414	415
209	416	417
210	418	419
211	420	421
212	422	423
213	424	425
214	426	427
215	428	429
216	430	431
217	432	433
218	434	435
219	436	437
220	438	439
221	440	441
222	442	443
223	444	445
224	446	447
225	448	449
226	450	451
227	452	453
228	454	455
229	456	457
230	458	459
231	460	461
232	462	463
233	464	465
234	466	467
235	468	469
236	470	471
237	472	473
238	474	475
239	476	477
240	478	479
241	480	481
242	482	483
243	484	485
244	486	487
245	488	489
246	490	491
247	492	493
248	494	495
249	496	497
250	498	499
251	500	501
252	502	503
253	504	505
254	506	507
255	508	509
256	510	511

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

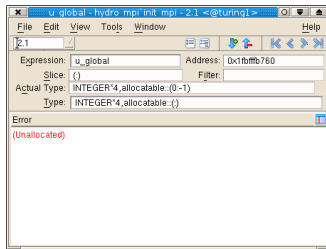


FIGURE 25 – u_global array

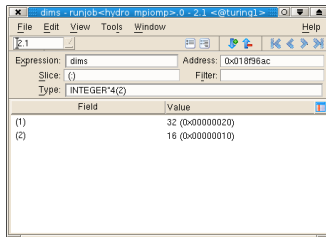


FIGURE 26 – Grid dimensions

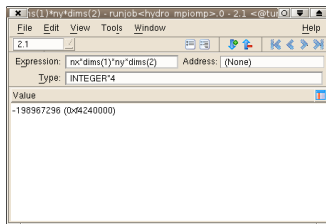
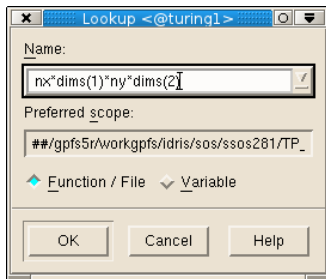


FIGURE 27 – A variable Lookup

- 1 Introduction to parallel debugging
- 2 Some principles of parallel debugging
- 3 Presentation of the HYDRO code
- 4 HPC debugging tools
- 5 Hands-on exercises
- 6 Large scale debugging
- 7 Conclusion

- *"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!**