Institut de
Mathématiques de
Bordeaux

The Distributed Debugging Tool (DDT)

13 février 2015

Khodor KHADRA, Ingénieur de Recherche Calcul Scientifique

Frequent errors when executing a program

- A program crash
  - Frequently a signal : segmentation fault, stack overflow, floating point exception, illegal instruction, …
  - Variety of causes : accessing memory out of range, infinite recursion, division by zero, overwritten stack, …
- Incorrect results
  - Many causes – no magic solutions : original dataset error, communication or synchronization problem, badlogic, beyond bound memory access or unexpected access, …
  - Intuition and brainpower required
- A deadlock or no progress
  - Application fails to terminate : infinite loops, deadlock (message ordering/matching issue, disagreement on collective MPI calls), …

Why using a debugging tool ?

- Printing variables inside the source code until the errors are detected costs a lot ;

- Using a debugging software allows :
  - To visualize the variables during execution
  - The knowledge of all the subprograms on which lines there is a problem
  - The use of breakpoints between specific lines to determine the region where there is a problem
  - Consequently to detect and correct the errors more easily

Compiling a code before debugging

- As always, when compiling the program that you wish to debug, you must add the debug flag to your compile command. For the most compilers this is -g.
- It is also advisable to turn off compiler optimizations as these can make debugging appear strange and unpredictable.

- NB : before executing the program, one can increase the size of the stack in order to analyze the segmentation errors : *ulimit -s unlimited*

Previous steps before using DDT

- Connecting to PLAFRIM:

```
ssh   -X   mygale
qsub   -IX   -qclustername   -lnodes=2:ppn=8   -lwalltime=08:00:00
```

  (if you do not use the -q option, you work on the nodes called fourmi)

- Loading the compiler and MPI libraries:

```
module   add   meta/intel-impi
```

- Compiling the source code:

```
mpif90   -g   -o   exec   file.f90
mpicc    -g    -o  exec    file.c
```

- Executing the program:

  `./exec` (sequential)

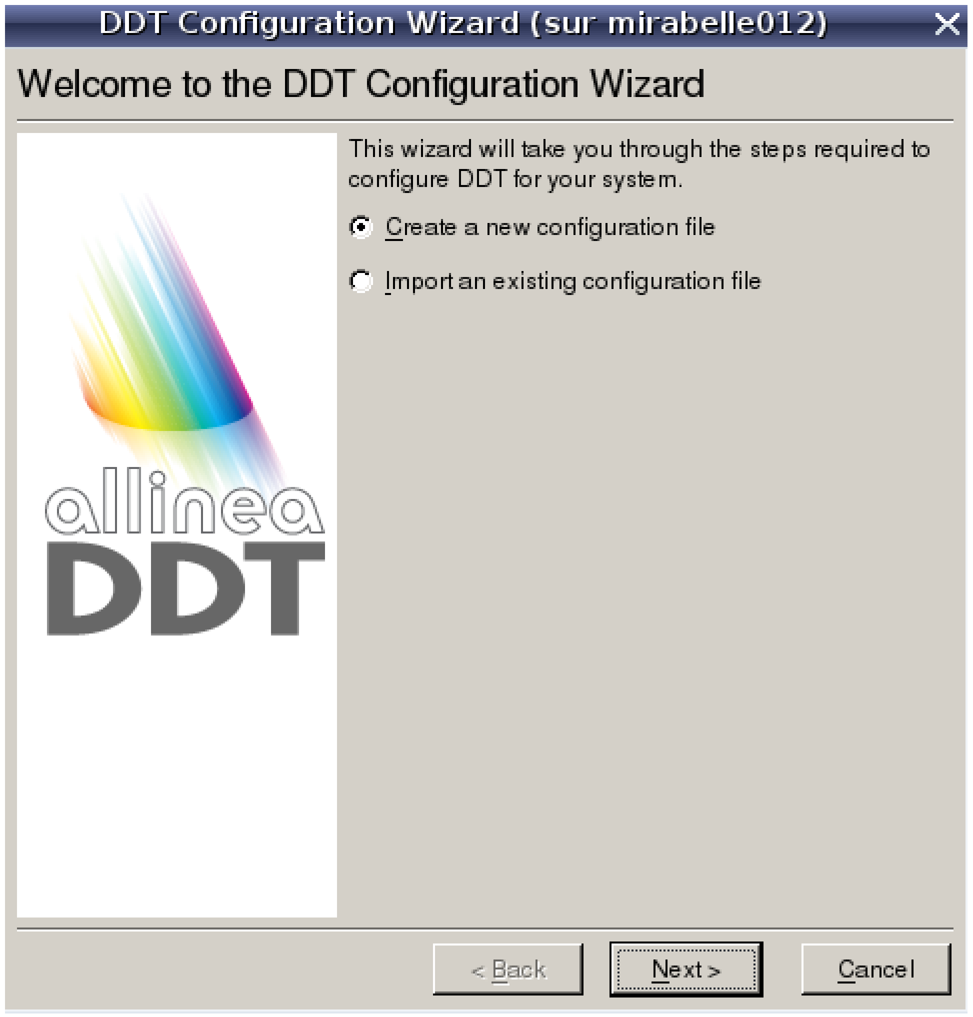
  `mpirun   -np   number_of_processes   ./exec` (parallel)

  One notices that there is a problem while executing

Configuration

- DDT has a Configuration Wizard to help simplify setting up DDT and choosing the correct options to start your programs. The first time you run DDT after installing it you will see that wizard

- Then click on Next and follow the simple instructions


DDT Configuration Wizard (sur mirabelle012)

Welcome to the DDT Configuration Wizard

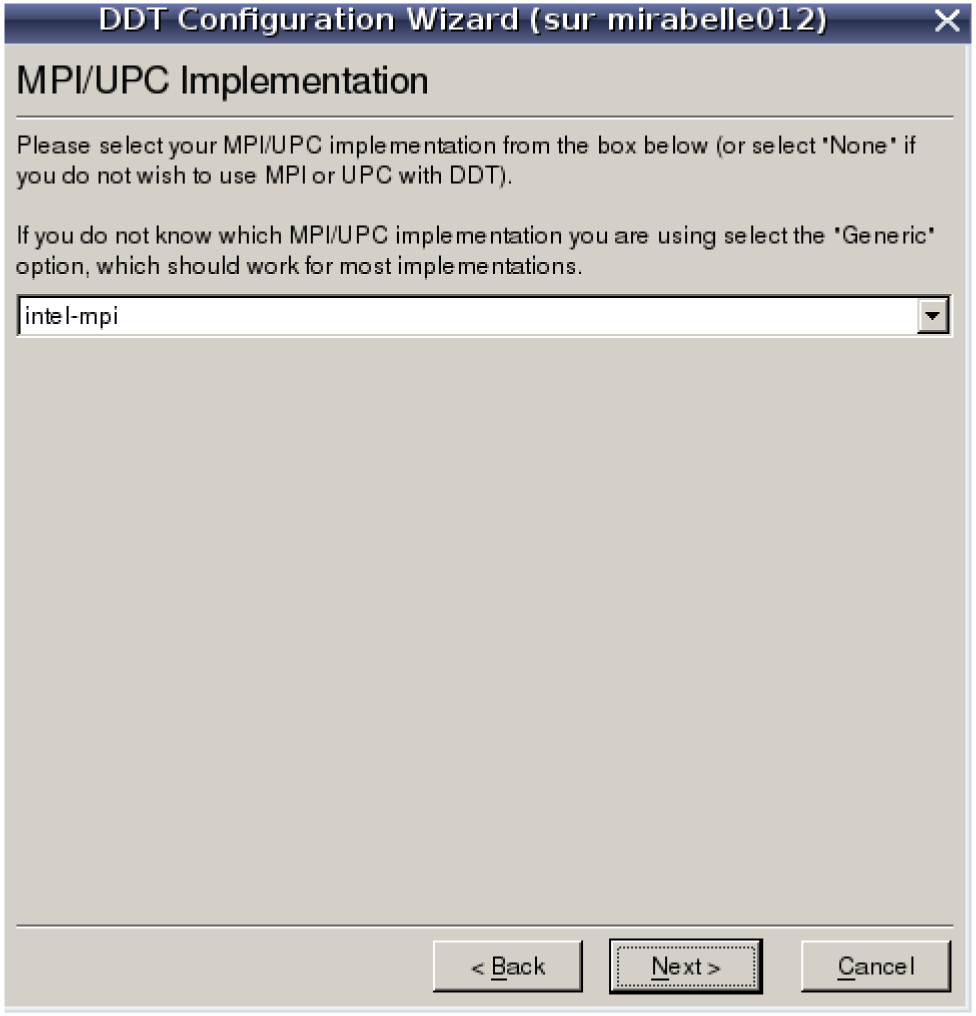
This wizard will take you through the steps required to configure DDT for your system.

[✓ Create a new configuration file]

Import an existing configuration file

Configuration

- After the welcome page you will see the MPI Implementation page.
- DDT will attempt to auto-detect and highlight your MPI implementation in the list, if this is not successful, select your MPI implementation manually.
- Once you have chosen or accepted an MPI Implementation, click on Next.


DDT Configuration Wizard (sur mirabelle012)

MPI/UPC Implementation

Please select your MPI/UPC implementation from the box below (or select 'None' if you do not wish to use MPI or UPC with DDT).

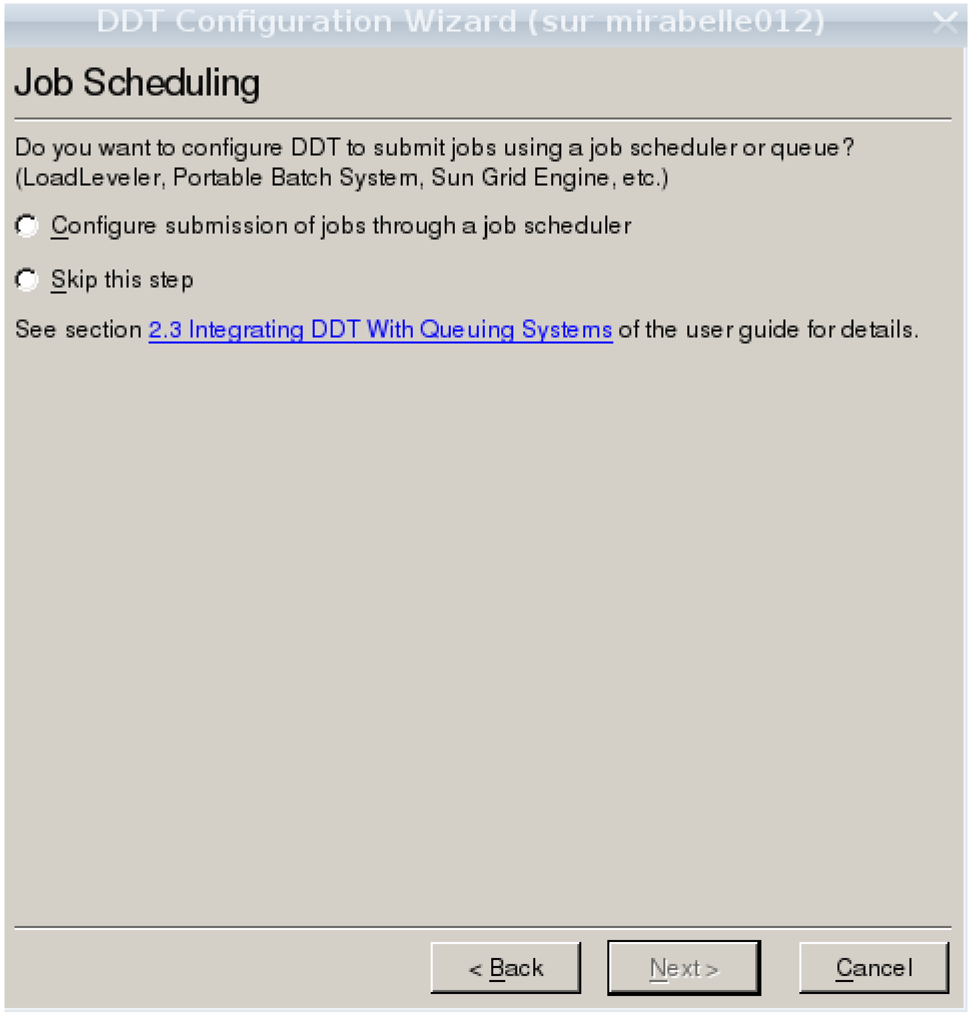
If you do not know which MPI/UPC implementation you are using select the 'Generic' option, which should work for most implementations.

intel-mpi

Configuration

- The Job Scheduling page asks if you want to submit your jobs using a job scheduler or queue.

- If you are using a job scheduler such as LoadLeveler, Portable Batch System or Sun Grid Engine select the Submit through a job scheduler option, otherwise skip this step.



Job Scheduling

Do you want to configure DDT to submit jobs using a job scheduler or queue?
(LoadLeveler, Portable Batch System, Sun Grid Engine, etc.)

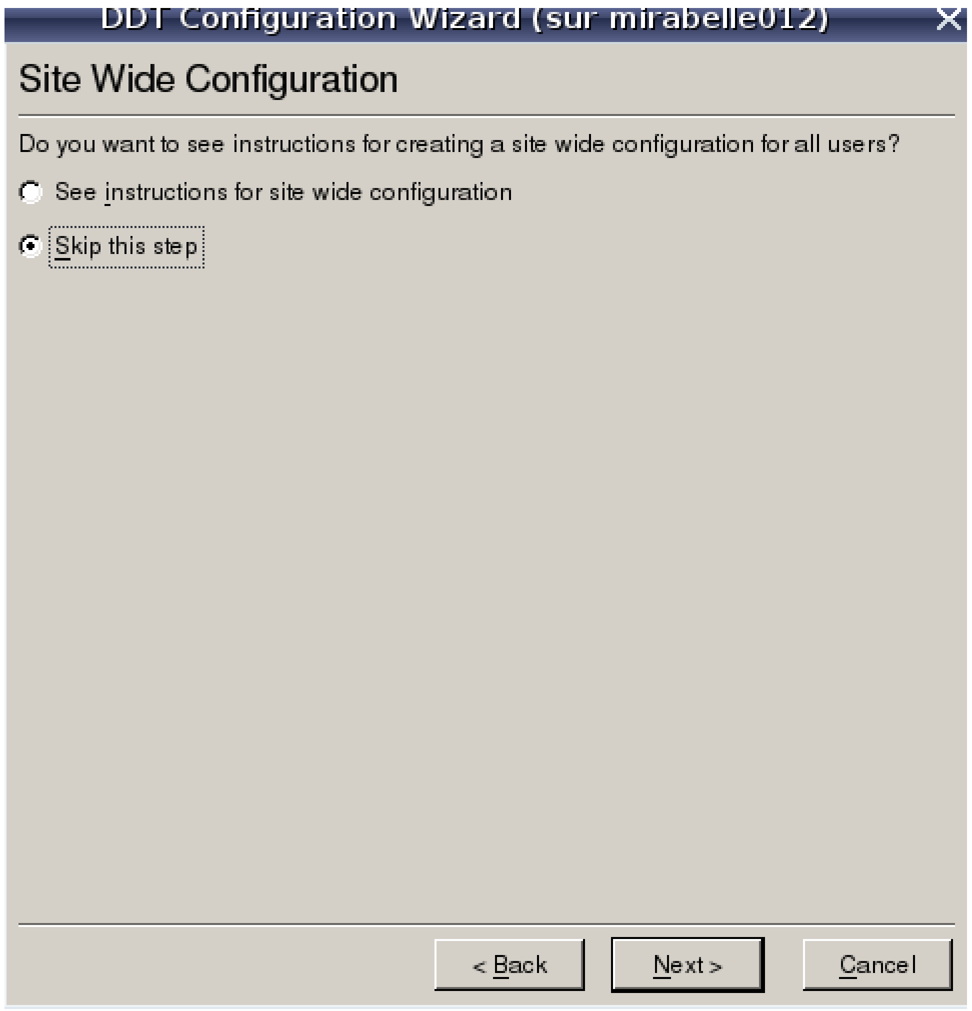
- ○ Configure submission of jobs through a job scheduler
- ○ Skip this step

See section 2.3 Integrating DDT With Queuing Systems of the user guide for details.

Configuration

- Side Wide Configuration

- You can skip this step.


DDT Configuration Wizard (sur mirabelle012)

Site Wide Configuration

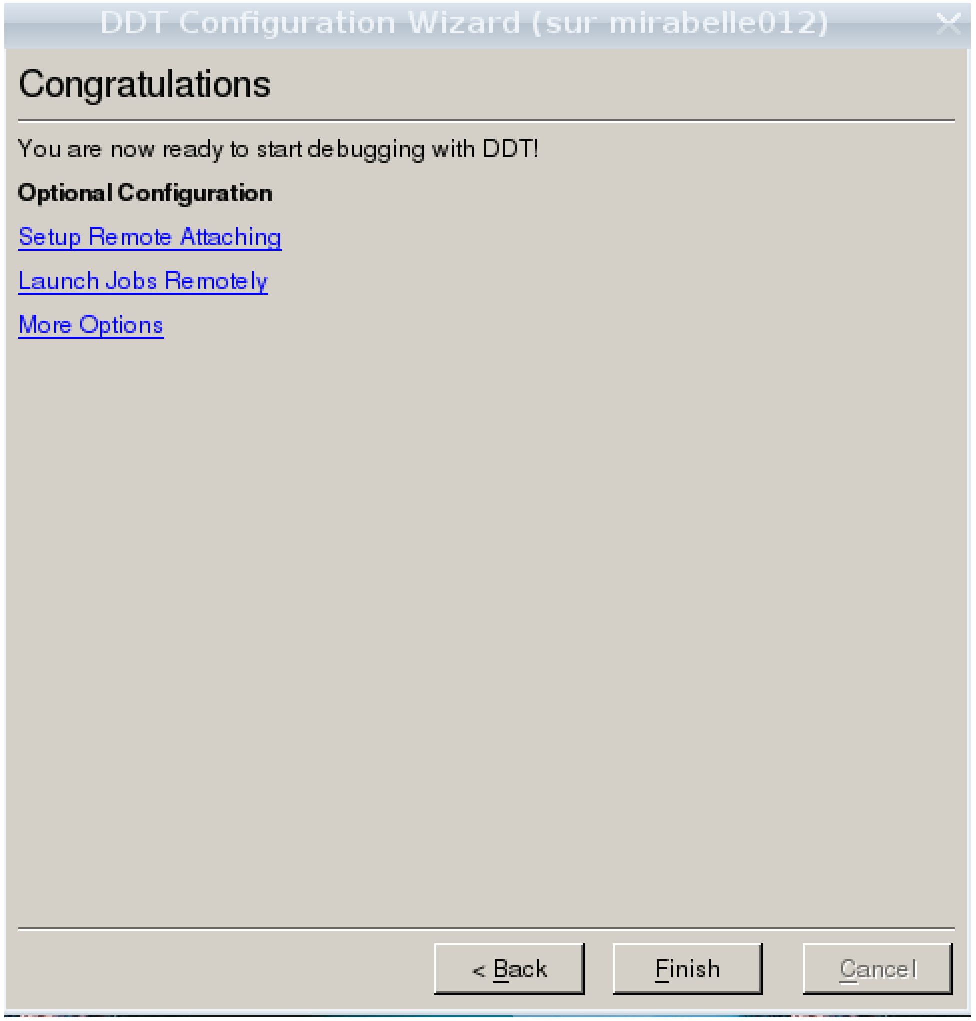
Do you want to see instructions for creating a site wide configuration for all users?

- See instructions for site wide configuration
- [✓ Skip this step]

Configuration

- The final congratulatory page contains links to other optional configuration settings. You can click on one of the hyperlinks to open the relevant options page or help file.

- Click on Finish to save these settings to the configuration file


DDT Configuration Wizard (sur mirabelle012)

Congratulations

You are now ready to start debugging with DDT!

Optional Configuration

Setup Remote Attaching

Launch Jobs Remotely

More Options

Recomendations of how working with DDT

STEP 1 :

Debugging without MPI, in a sequential mode, as much as you can all the errors which are not provided by MPI instructions.

- On your xterm:

  ```
  ./exec
  ```
  Notice the kind of error

- Open DDT without MPI. DDT perfoms its debugging from the “program main” until the “end program main”

- Correct in the source code, re-compile and re-execute

STEP 2 :

Debugging with MPI, in a parallel mode, by choosing the number of processes.

- On your xterm:

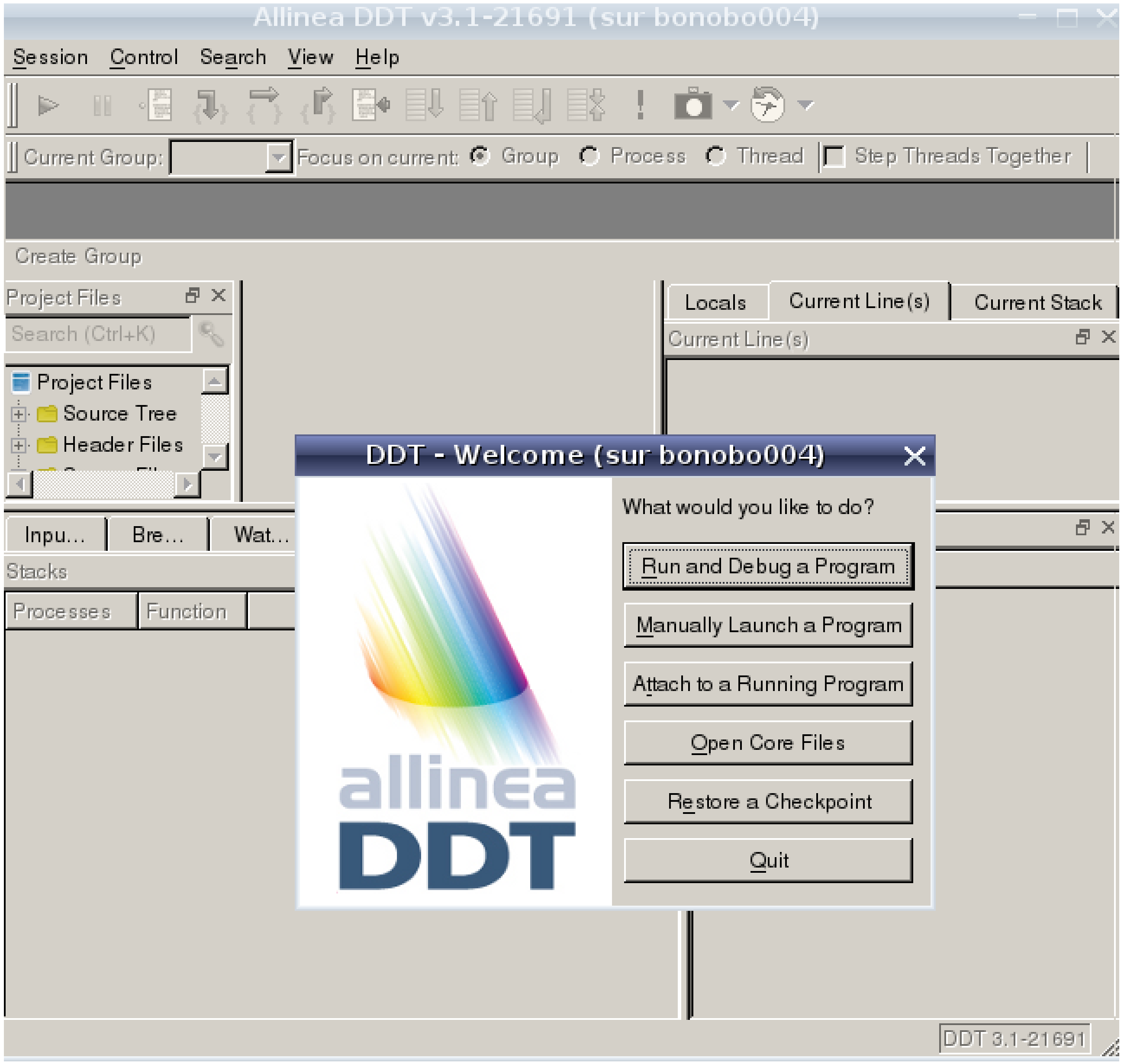
  ```
  mpirun -np number_of_processes ./exec
  ```
  Notice the kind of error

- Open DDT with MPI. DDT perfoms its debugging between the MPI INITialization instructions and the MPI_FINALIZE instruction.

- Correct in the source code, re-compile and re-execute

Starting DDT

- Once DDT has started it will display the Welcome Screen.

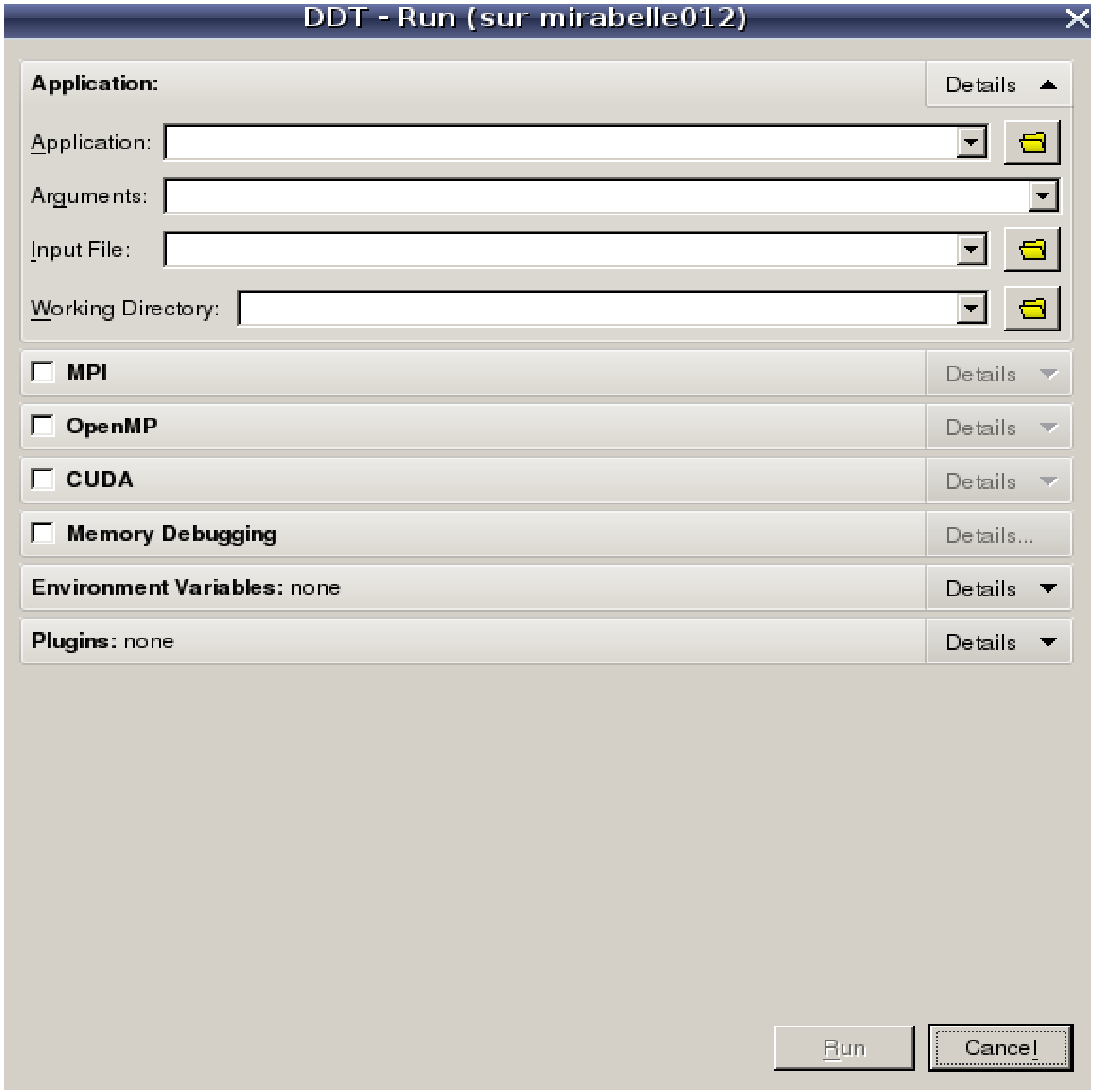


Running and debugging a program

- The Welcome Screen allows you to choose what kind of debugging you want to do. You can:
  - run a program from DDT and debug it
  - debug a program you launch manually (e.g. on the command line)
  - attach to an already running program
  - open core files generated by a program that crashed
  - restore a checkpoint of a program and continue debugging

Running and debugging a program

- If you click the Run and Debug a Program button on the Welcome Screen you will see that window
- In the Application field, you write the path to your binary executable file
- If you click on MPI, you enter the number of processes that you wish to run and you click on button Run.


DDT - Run (sur mirabelle012)

Application:

Application:

Arguments:

Input File:

Working Directory:

☐ MPI

☐ OpenMP

☐ CUDA

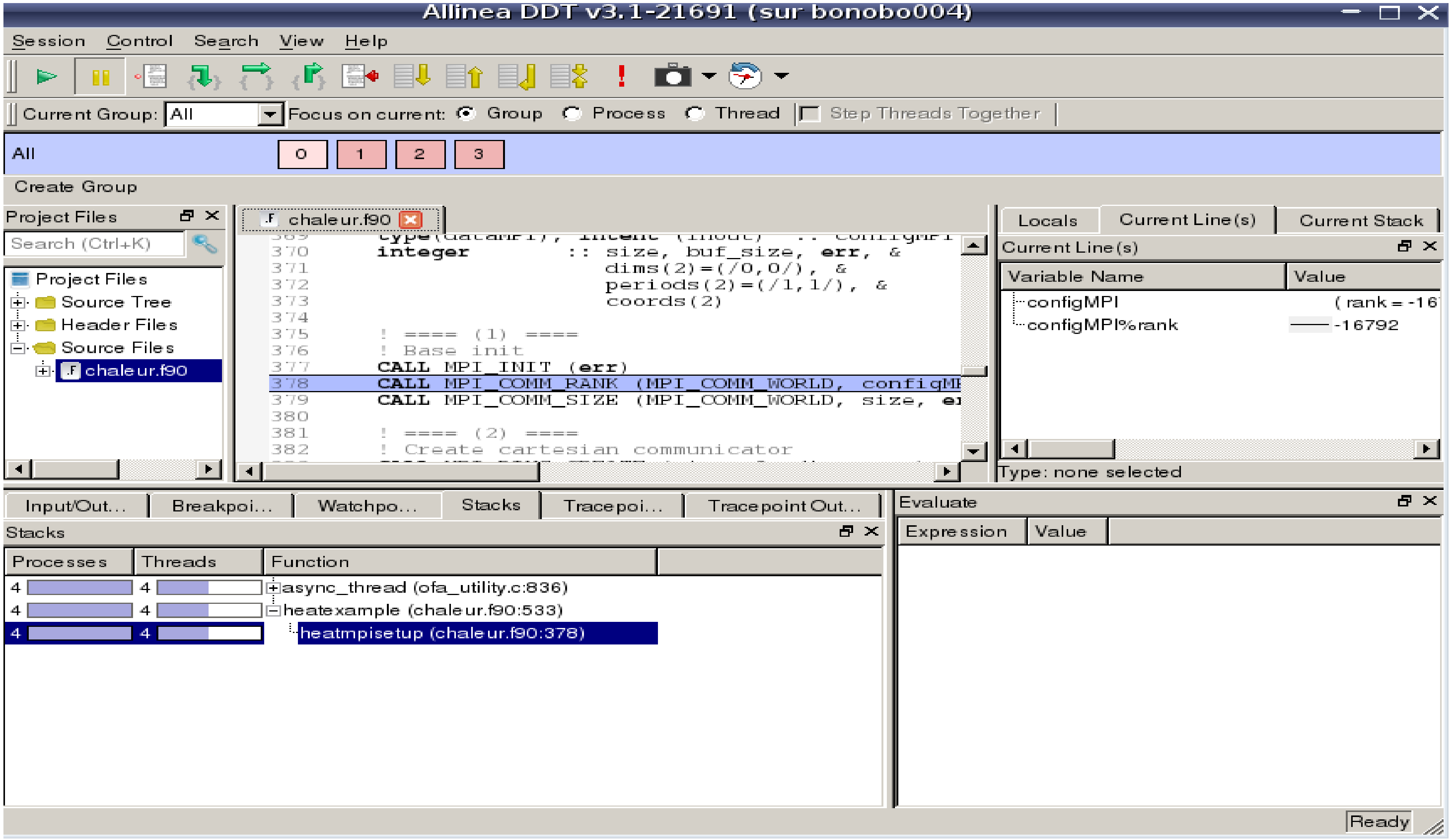
☐ Memory Debugging

Environment Variables: none

Plugins: none

Running and debugging a program

- This is the screen you see when for example you load DDT with MPI. When an MPI initialization instruction is highlighted, go on while pressing on the green upleft arrow.


Allinea DDT v3.1-21691 (sur bonobo004)
Session Control Search View Help
Current Group: All
Focus on current: Group Process Thread
Step Threads Together
All 0 1 2 3
Create Group
Project Files
Search (Ctrl+K)
Project Files
Source Tree
Header Files
Source Files
chaleur.f90
Locals Current Line(s) Current Stack
Type: none selected
Input/Out... Breakpoi... Watchpo... Stacks Tracepoi... Tracepoint Out...
Evaluate
Expression Value
Ready

```
370        integer           :: size, buf_size, err, &
371                              dims(2)=(/0,0/), &
372                              periods(2)=(/1,1/), &
373                              coords(2)
374
375        ! ==== (1) ====
376        ! Base init
377        CALL MPI_INIT (err)
378        CALL MPI_COMM_RANK (MPI_COMM_WORLD, configMP
379        CALL MPI_COMM_SIZE (MPI_COMM_WORLD, size, e
380
381        ! ==== (2) ====
382        ! Create cartesian communicator
```

Current Line(s)

Variable Name	Value
configMPI	(rank = -16
configMPI%rank	-16792

Stacks

Processes	Threads	Function
4	4	async_thread (ofa_utility.c:836)
4	4	heatexample (chaleur.f90:533)
4	4	heatmpisetup (chaleur.f90:378)

Some basic functionalities of DDT

- Using the Source Code Viewer, locate the position in your code that you want to place breakpoints at specific lines to make a break in your program, and print variables at this stage;

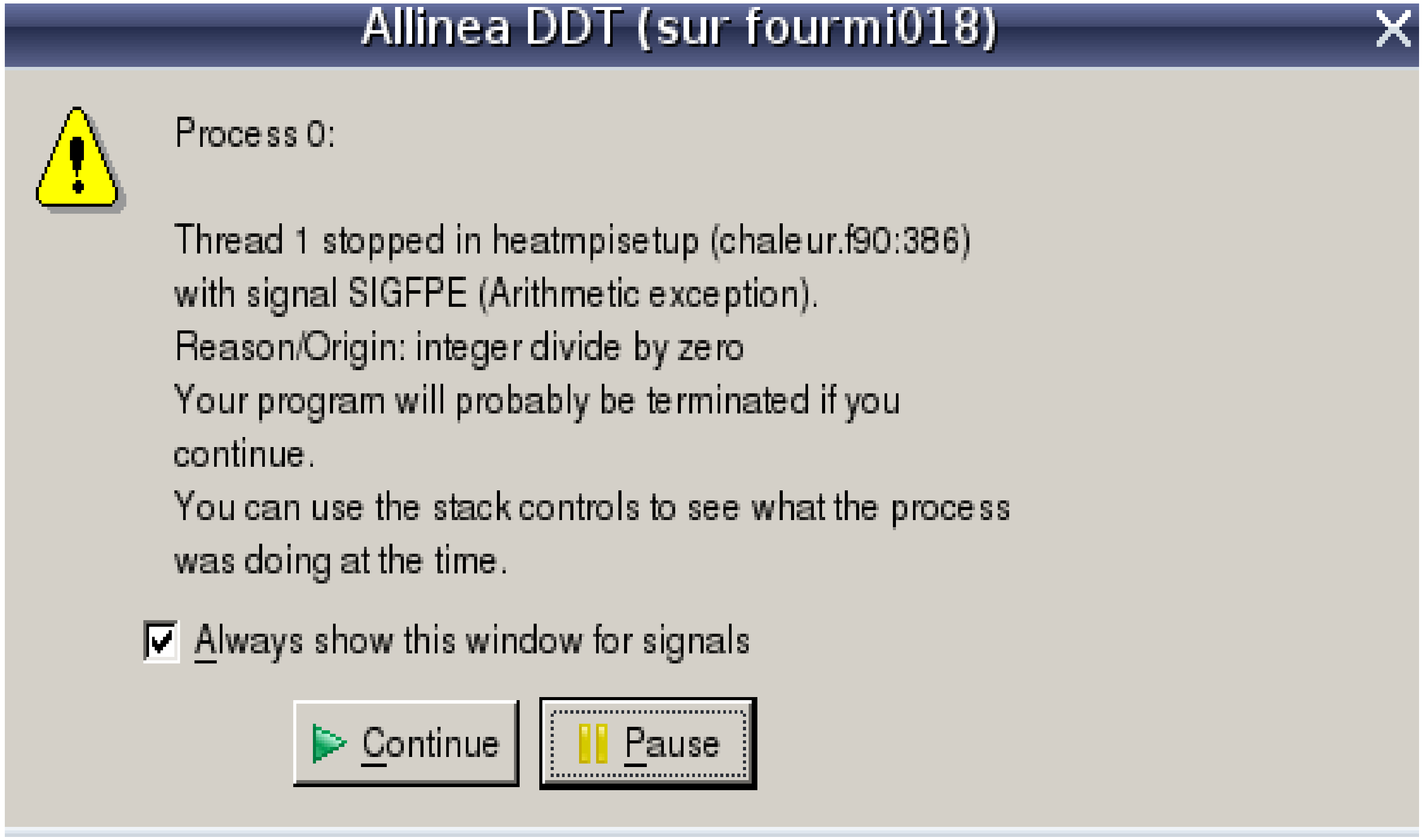
- Using tracepoints which allow to see what lines of code your program is executing – and the variables without stopping it.

- When right clicking on an array variable in the Source Code Viewer, one can have many informations by viewing the data array variable and across processors.

- When clicking on « Stacks », one can see all the different calls of subprograms where the errors occur

Floating Point Exception

- Identify the line of the arithmetic exception (integer divide by zero)



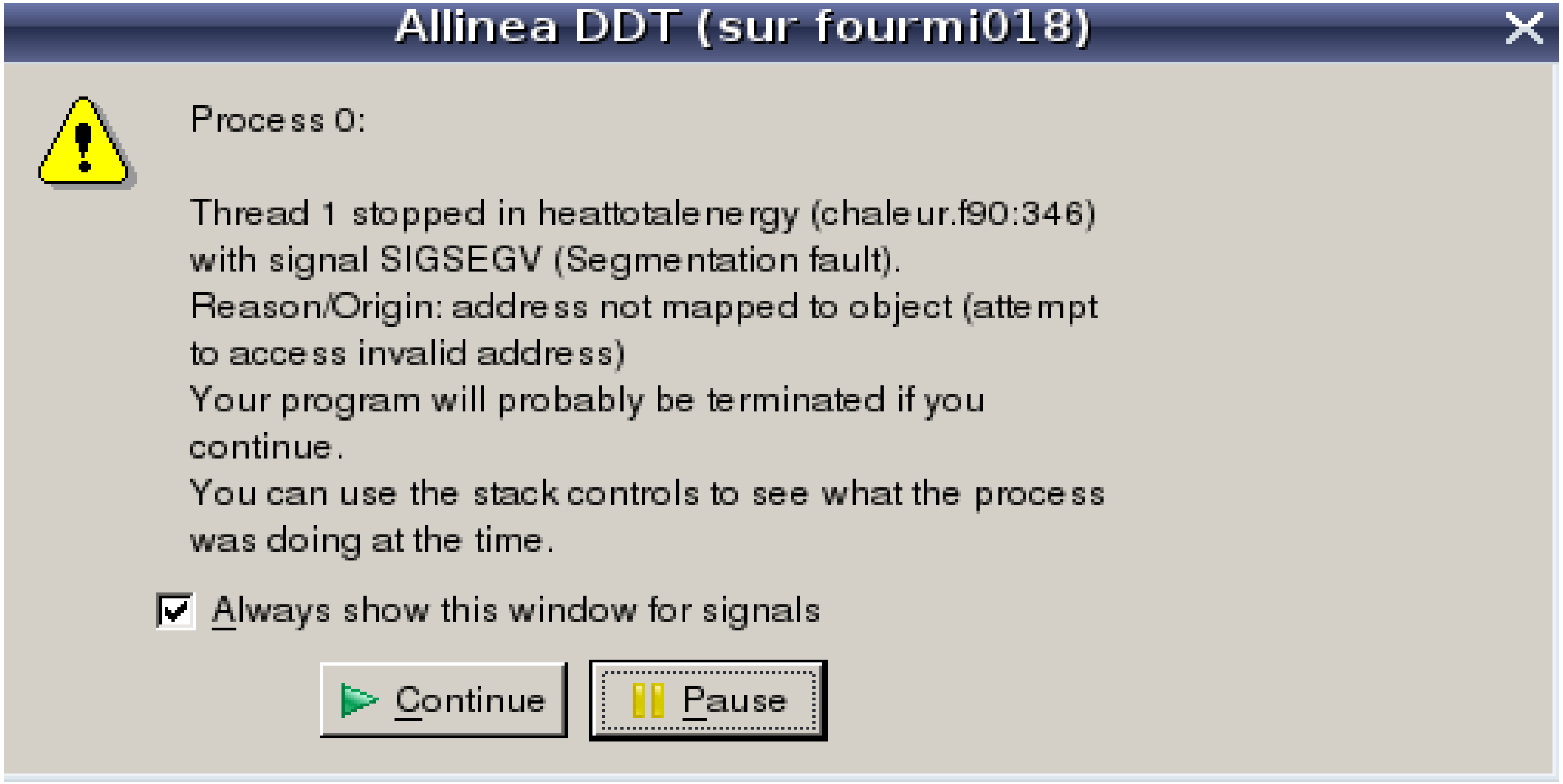
Process 0:

Thread 1 stopped in heatmpisetup (chaleur.f90:386)
with signal SIGFPE (Arithmetic exception).
Reason/Origin: integer divide by zero
Your program will probably be terminated if you
continue.
You can use the stack controls to see what the process
was doing at the time.

☑ Always show this window for signals

Segmentation Fault

- Identify the line where the variable is not allocated or is out of memory acces



```
Process 0:

Thread 1 stopped in heattotalenergy (chaleur.f90:346)
with signal SIGSEGV (Segmentation fault).
Reason/Origin: address not mapped to object (attempt
to access invalid address)
Your program will probably be terminated if you
continue.
You can use the stack controls to see what the process
was doing at the time.
```

[✓] Always show this window for signals

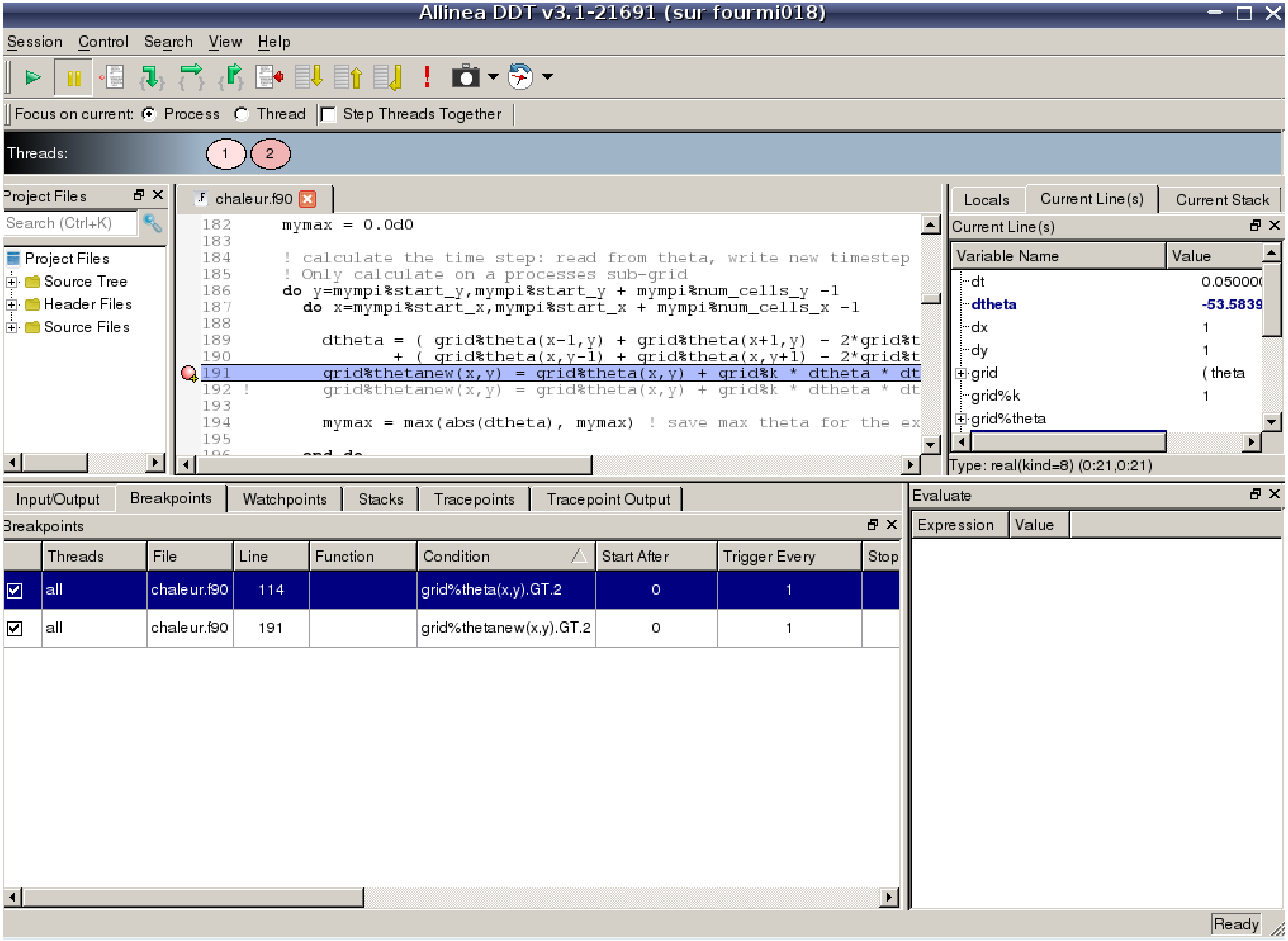
Arithmetic Error (Not A Number)

- NaN (Not a Number), is a numeric data type value representing an undefined or unrepresentable value.
- Notice that if you run the program, it will end without being interrupted though it provides NaN values.
- To solve this kind of problem, it is important to know how your program works in order to :
  - Identify the variables (arrays) where the NaN occur;
  - Identify the regions between which lines the values the variables increase abnormally;
  - Use breakpoints and conditional breakpoints on the lines where you suspect the problem occurs;
  - Detect and correct the error.
- Advice: you can use Multi-Dimensional Array (MDA) Viewer to print easily the values of an array

Breakpoints

- The use of breakpoints on specific lines in the source code allows to execute the program until that line, to print the variables at this stage and across processes.
- Using the Source Code Viewer to add a breakpoint
  - first locate the position in your code that you want to place a breakpoint at
  - every breakpoint is listed under the breakpoints tab towards the bottom of DDT's window.
- Conditional Breakpoints
  - Select the breakpoints tab to view all the breakpoints in your program. You may add a condition to any of them by clicking on the condition cell in the breakpoint table and entering an expression that evaluates to true or false.
  - The expression should be in the same language as your program.
  - Each time a process (in the group the breakpoint is set for) passes this breakpoint it will evaluate the condition and break only if it returns true.
- Deleting a Breakpoint
  - Breakpoints are deleted by either right-clicking on the breakpoint in the breakpoints panel, or by rightclicking at the file/line of the breakpoint.

Breakpoints


Allinea DDT v3.1-21691 (sur fourmi018)
Session Control Search View Help
Focus on current: Process Thread Step Threads Together
Threads: 1 2
Project Files
Search (Ctrl+K)
Project Files
Source Tree
Header Files
Source Files
chaleur.f90
Locals Current Line(s) Current Stack
Current Line(s)
Input/Output Breakpoints Watchpoints Stacks Tracepoints Tracepoint Output
Breakpoints
Evaluate
Expression Value
Ready

```
182      mymax = 0.0d0
183
184      ! calculate the time step: read from theta, write new timestep
185      ! Only calculate on a processes sub-grid
186      do y=mympi%start_y,mympi%start_y + mympi%num_cells_y -1
187        do x=mympi%start_x,mympi%start_x + mympi%num_cells_x -1
188
189          dtheta = ( grid%theta(x-1,y) + grid%theta(x+1,y) - 2*grid%t
190                 + ( grid%theta(x,y-1) + grid%theta(x,y+1) - 2*grid%t
191          grid%thetanew(x,y) = grid%theta(x,y) + grid%k * dtheta * dt
192 !        grid%thetanew(x,y) = grid%theta(x,y) + grid%k * dtheta * dt
193
194          mymax = max(abs(dtheta), mymax) ! save max theta for the ex
195
```

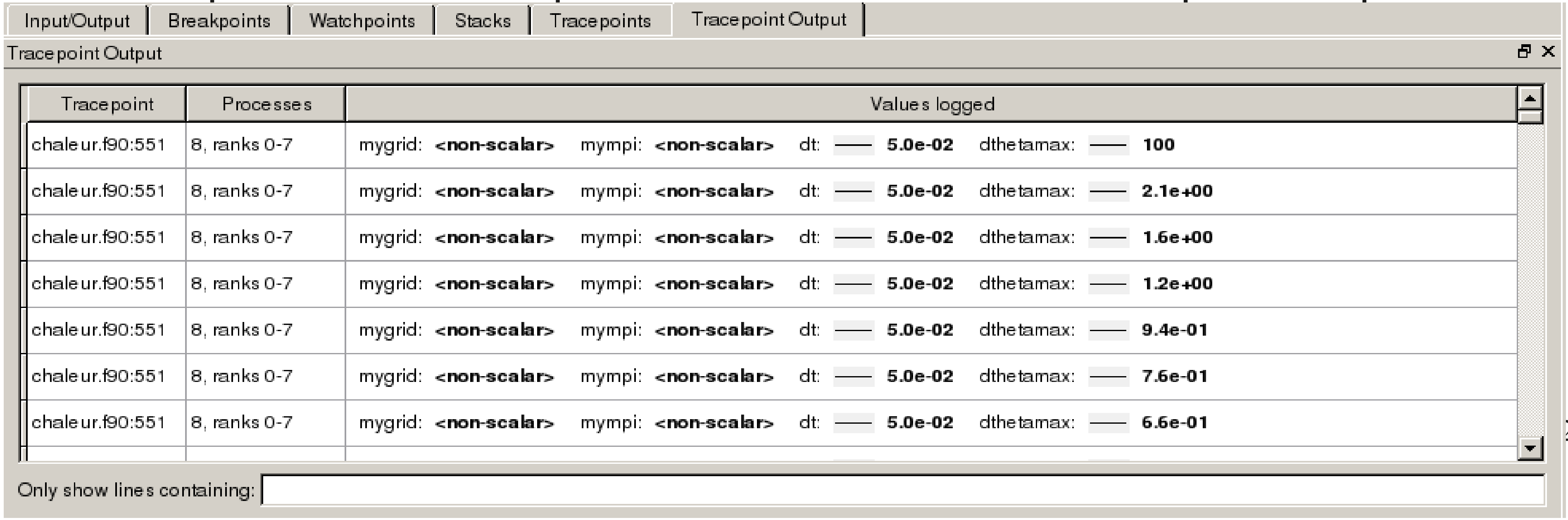
Variable Name	Value
dt	0.05000(
dtheta	**-53.5839**
dx	1
dy	1
grid	(theta
grid%k	1
grid%theta	

Type: real(kind=8) (0:21,0:21)

	Threads	File	Line	Function	Condition	Start After	Trigger Every	Stop
☑	all	chaleur.f90	114		grid%theta(x,y).GT.2	0	1	
☑	all	chaleur.f90	191		grid%thetanew(x,y).GT.2	0	1	

Tracepoints

- Tracepoints allow you to see what lines of code your program is executing – and the variables – without stopping it. Whenever a thread reaches a tracepoint it will print the file and line number of the tracepoint to the Input/Output view. You can also capture the value of any number of variables or expressions at that point.
- Setting a tracepoint
  - Tracepoints are added by either right-clicking on a line in the Source Code Viewer and selecting the Add Tracepoint menu item. In that case, a number of variables based on the current line of code will be captures by default.
- Tracepoint Output
  - The output from the tracepoints can be found in the Tracepoint Output view.



Input/Output | Breakpoints | Watchpoints | Stacks | Tracepoints | Tracepoint Output

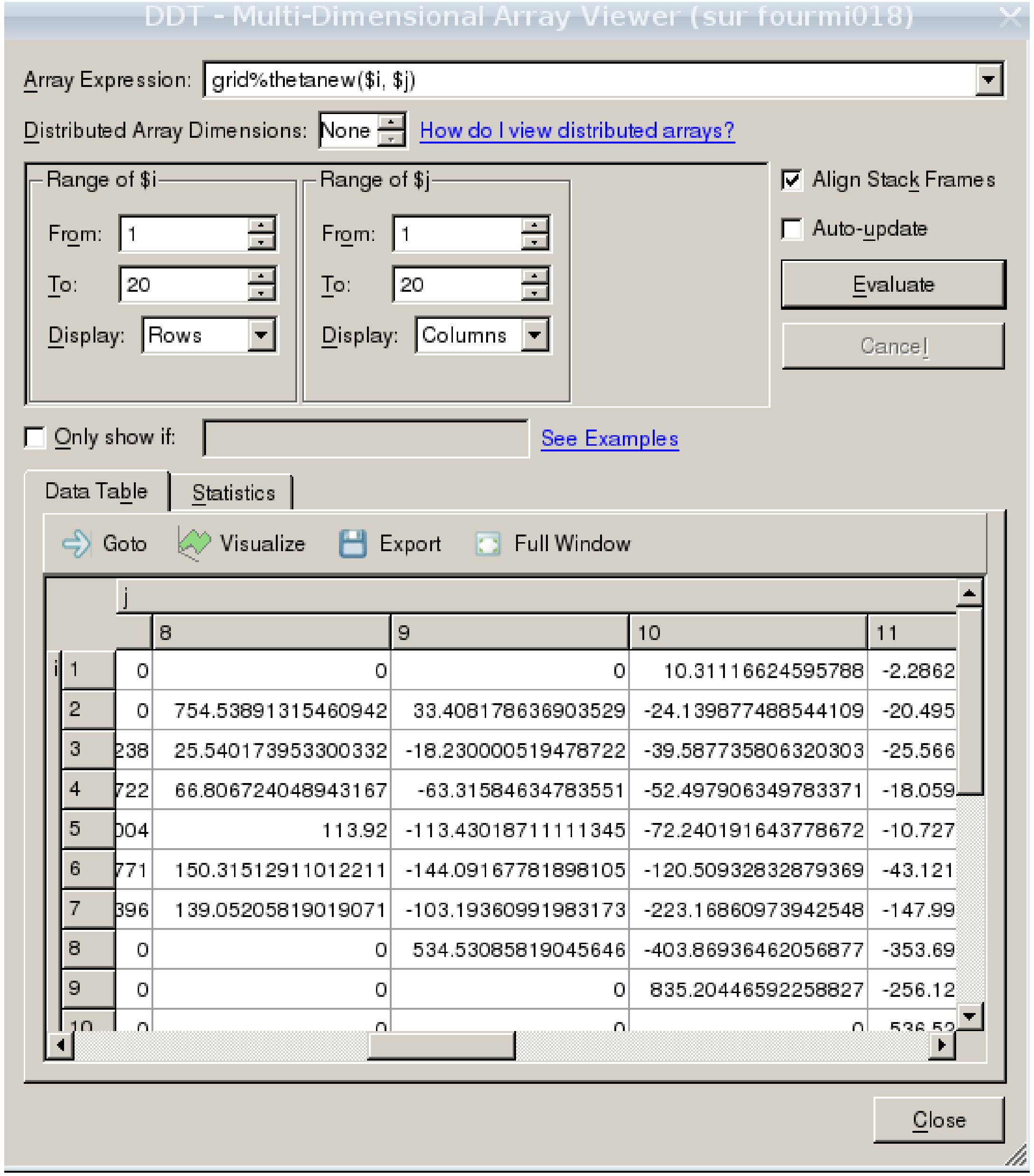
Tracepoint Output

Tracepoint	Processes	Values logged
chaleur.f90:551	8, ranks 0-7	mygrid: **<non-scalar>** mympi: **<non-scalar>** dt: **5.0e-02** dthetamax: **100**
chaleur.f90:551	8, ranks 0-7	mygrid: **<non-scalar>** mympi: **<non-scalar>** dt: **5.0e-02** dthetamax: **2.1e+00**
chaleur.f90:551	8, ranks 0-7	mygrid: **<non-scalar>** mympi: **<non-scalar>** dt: **5.0e-02** dthetamax: **1.6e+00**
chaleur.f90:551	8, ranks 0-7	mygrid: **<non-scalar>** mympi: **<non-scalar>** dt: **5.0e-02** dthetamax: **1.2e+00**
chaleur.f90:551	8, ranks 0-7	mygrid: **<non-scalar>** mympi: **<non-scalar>** dt: **5.0e-02** dthetamax: **9.4e-01**
chaleur.f90:551	8, ranks 0-7	mygrid: **<non-scalar>** mympi: **<non-scalar>** dt: **5.0e-02** dthetamax: **7.6e-01**
chaleur.f90:551	8, ranks 0-7	mygrid: **<non-scalar>** mympi: **<non-scalar>** dt: **5.0e-02** dthetamax: **6.6e-01**

Only show lines containing:

Multi-Dimensional Array Viewer (MDA)

- To open MDA, right-click on a variable in the Source Code or open it directly by selecting the Multi-Dimensional Array Viewer menu item from the View menu.
- The Array Expression is an expression containing a number of subscript metavariables that are substituted with the subscripts of the array. For example, the expression myArray($i, $j) has 2 metavariables, $i and $j. The metavariables are unrelated to the variables in your program.
- You define the range of each metavariable. The Array Expression is evaluated for each combination of $i , $j , etc. and the results are shown in the Data Table (Click on Evaluate). You may want the Data Table to only show elements that fit a certain criteria (Only show if).


DDT - Multi-Dimensional Array Viewer (sur fourmi018)

Array Expression: grid%thetanew($i, $j)

Distributed Array Dimensions: None

How do I view distributed arrays?

Range of $i		Range of $j	
From:	1	From:	1
To:	20	To:	20
Display:	Rows	Display:	Columns

[✓] Align Stack Frames

[ ] Auto-update

[ ] Only show if:

See Examples

i \ j		8	9	10	11
1	0	0	0	10.31116624595788	-2.2862
2	0	754.53891315460942	33.408178636903529	-24.139877488544109	-20.495
3	238	25.540173953300332	-18.230000519478722	-39.587735806320303	-25.566
4	722	66.806724048943167	-63.31584634783551	-52.497906349783371	-18.059
5	004	113.92	-113.43018711111345	-72.240191643778672	-10.727
6	771	150.31512911012211	-144.09167781898105	-120.50932832879369	-43.121
7	396	139.05205819019071	-103.19360991983173	-223.1686097394548	-147.99
8	0	0	534.53085819045646	-403.86936462056877	-353.69
9	0	0	0	835.20446592258827	-256.12
10	0	0	0	0	536.52

Deadlock

- A deadlock is a situation wherein two or more competing actions are each waiting for the other to finish, and thus neither ever does.
- In an operating system, a deadlock is a situation which occurs when a process enters a waiting state because a resource requested by it is being held by another waiting process, which in turn is waiting for another resource. If a process is unable to change its state indefinitely because the resources requested by it are being used by other waiting process, then the system is said to be in a deadlock.
- Deadlock is a common problem in multiprocessing systems, parallel computing and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization.
- Frequent Deadlock situation occurs when using blocking point-to-point MPI routines: a process executes a receive routine and there is no corresponding call to a send routine.

Message Queues

- One can use DDT to detect common errors such as deadlock, where all processes are waiting for each other, or for detecting when messages are present that are unexpected, which can correspond to two processes disagreeing about the state of progress through a program.

- DDT's Message Queue debugging feature shows the status of the internal message buffers of MPI, for example showing the messages that have been sent by a process but not yet received by the target.

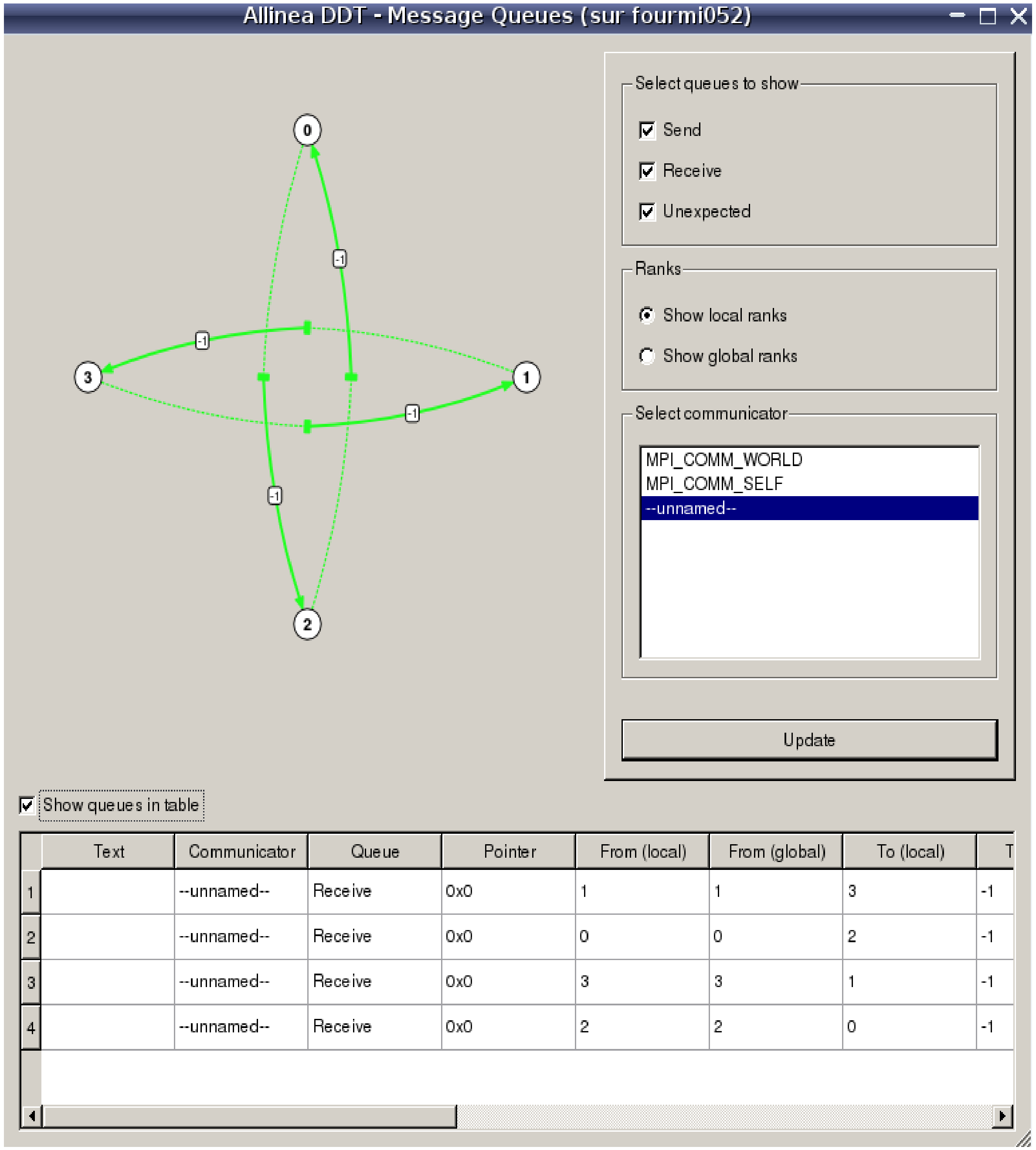
- Open the Message Queues window from the View menu.

Message Queues

- There are three different types of message queues about which there is information. Different colours are used to display messages from each type of queue.
- Send Queue : Calls to MPI send functions that have not yet completed.
- Receive Queue : Calls to MPI receive functions that have not yet completed.
- Unexpected Message Queue : Represents messages received by the system but the corresponding receive function call has not yet been made.
- Messages in the Send queue are represented by a red arrow, pointing from the sender to the recipient. The line is solid on the sender side, but dashed on the received side (to represent a message that has been Sent but not yet been Received).
- Messages in the Receive queue are represented by a green arrow, pointing from the sender to the recipient. The line is dashed on the sender side, but solid on the recipient side (to represent the recipient being ready to receive a message that has not yet been sent).
- Messages in the Unexpected queue are represented by a dashed blue arrow, pointing from sender of the unexpected message to the recipient.

Message Queues

- If you see a loop in the graph, it can be because of deadlock – every process waiting to receive from the preceding process in the loop.



Select queues to show
[✓] Send
[✓] Receive
[✓] Unexpected

Ranks
[✓ Show local ranks]
Show global ranks

Select communicator
MPI_COMM_WORLD
MPI_COMM_SELF
[✓ --unnamed--]

[✓] Show queues in table

	Text	Communicator	Queue	Pointer	From (local)	From (global)	To (local)	T
1		--unnamed--	Receive	0x0	1	1	3	-1
2		--unnamed--	Receive	0x0	0	0	2	-1
3		--unnamed--	Receive	0x0	3	3	1	-1
4		--unnamed--	Receive	0x0	2	2	0	-1

Memory Management

- The good and optimal memory management in a program is essential.
- Good reactions to handle with :
  - Check the memory that your program requires during execution.
  - Check that you do not deallocate arrays before allocating them, or that you do not compute with arrays after deallocating them.
  - Check that you systematically deallocate arrays once u have allocated them, and finished to compute with, despite it may not cause a damage or errors during the execution.
  - Check that if X Gb are allocated in your arrays at the beginning of the program, X Gb are deallocated at the end.
  - Check the memory usage on each process
  - Check for example in an iterative method, at each step, that the memory is not increasing.

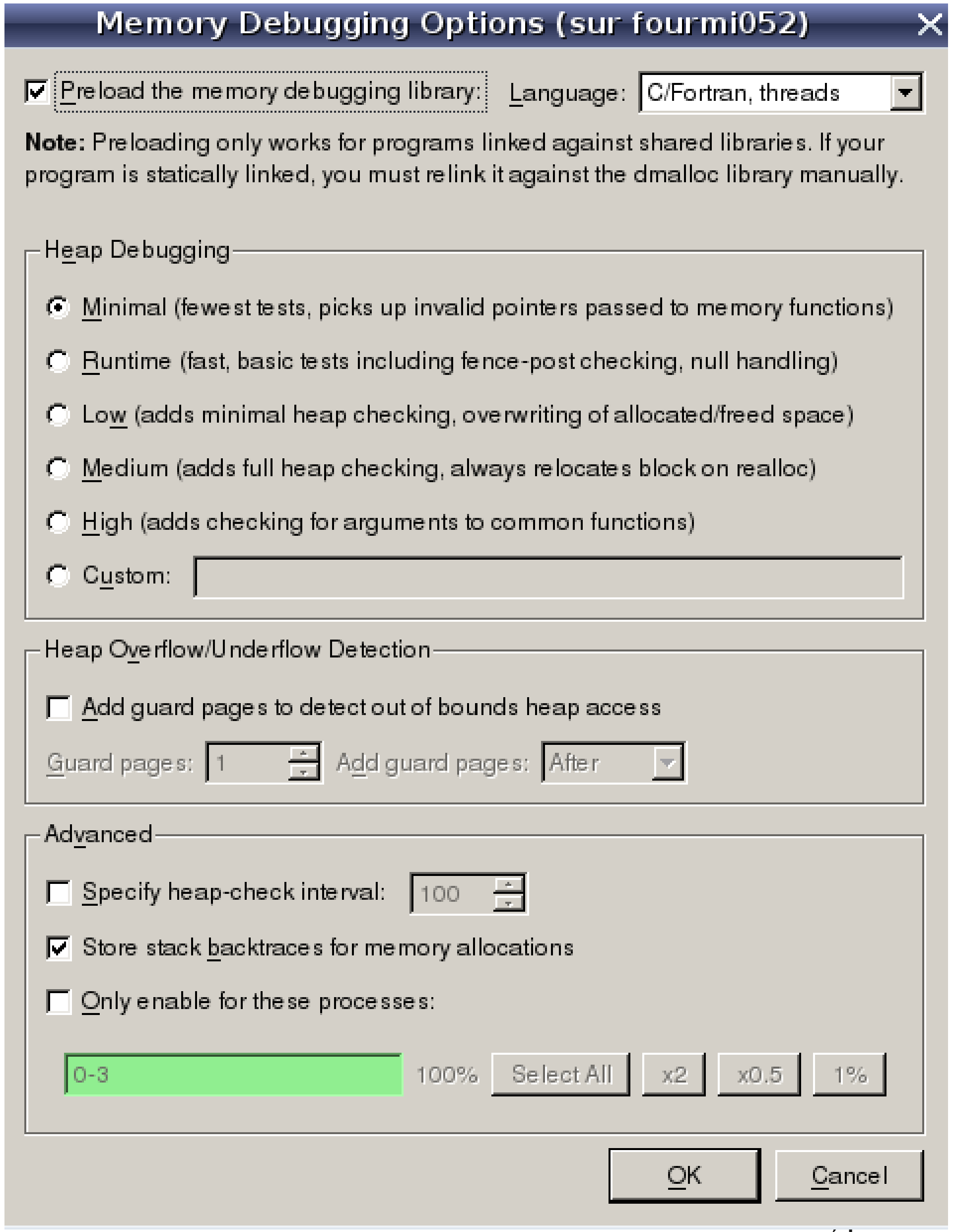
Memory debugging

- Allinea DDT has a powerful parallel memory debugging capability. This feature intercepts calls to the system memory allocation library, recording memory usage and monitoring correct usage of the library by performing heap and bounds checking.
- Typical problems that can be resolved by using Allinea DDT with memory debugging enabled include:
  - Memory exhaustion due to memory leaks can be prevented by examining the Current Memory Usage display which groups and quantifies memory according to the location at which blocks have been allocated.
  - Persistent but random crashes caused by access to memory beyond the bounds of an allocation block – can be resolved by using the Guard Pages feature
  - Crashing due to deallocation of the same memory block twice and other forms deallocation of an invalid pointers – for example deallocating a pointer that is not at the start of an allocation.

Memory Debugging

Enabling Memory Debugging from the Run window : click on the Memory Debugging checkbox.

- The two most significant options are:
  - Preload the memory debugging library
  - The box showing C/Fortran, No Threads in the screen shot – click here and select the option that best matches your program, be it C/Fortran, C++, Single-Threaded, Multi-Threaded.
- The Heap Debugging section allows you to turn on/off specific memory debugging features.
  - Minimal will catch trivial memory errors such as deallocating memory twice.
- You can turn on Heap Overflow/Underflow Detection to detect out of bounds heap access.


Memory Debugging Options (sur fourmi052)

[✓] Preload the memory debugging library: Language: C/Fortran, threads

Note: Preloading only works for programs linked against shared libraries. If your program is statically linked, you must relink it against the dmalloc library manually.

Heap Debugging
- [●] Minimal (fewest tests, picks up invalid pointers passed to memory functions)
- ○ Runtime (fast, basic tests including fence-post checking, null handling)
- ○ Low (adds minimal heap checking, overwriting of allocated/freed space)
- ○ Medium (adds full heap checking, always relocates block on realloc)
- ○ High (adds checking for arguments to common functions)
- ○ Custom:

Heap Overflow/Underflow Detection
- [ ] Add guard pages to detect out of bounds heap access

Guard pages: 1 Add guard pages: After

Advanced
- [ ] Specify heap-check interval: 100
- [✓] Store stack backtraces for memory allocations
- [ ] Only enable for these processes:

0-3 100% Select All x2 x0.5 1%

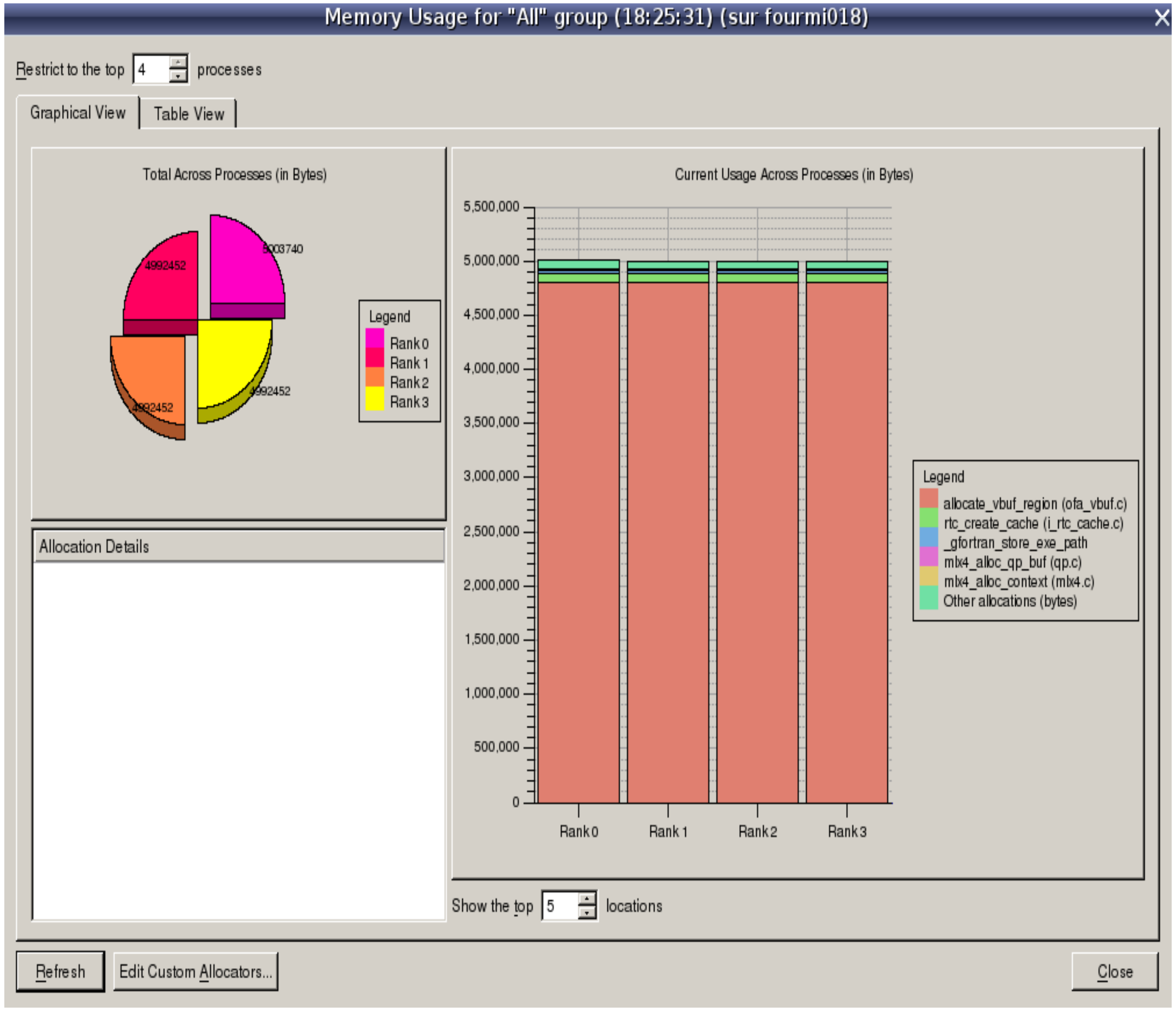
OK Cancel

Memory Debugging

Current Memory Usage

- Memory leaks can be a significant problem for software developers.
  If your application's memory usage grows faster than expected or continues to grow through its execution then it is possible that memory is being allocated which is not being returned when it is no longer required.

- At any point in your program when there is a breakpoint you can go to View → Current Memory Usage and DDT will then display the currently allocated memory in your program for the currently selected process group.


Memory Usage for "All" group (18:25:31) (sur fourmi018)

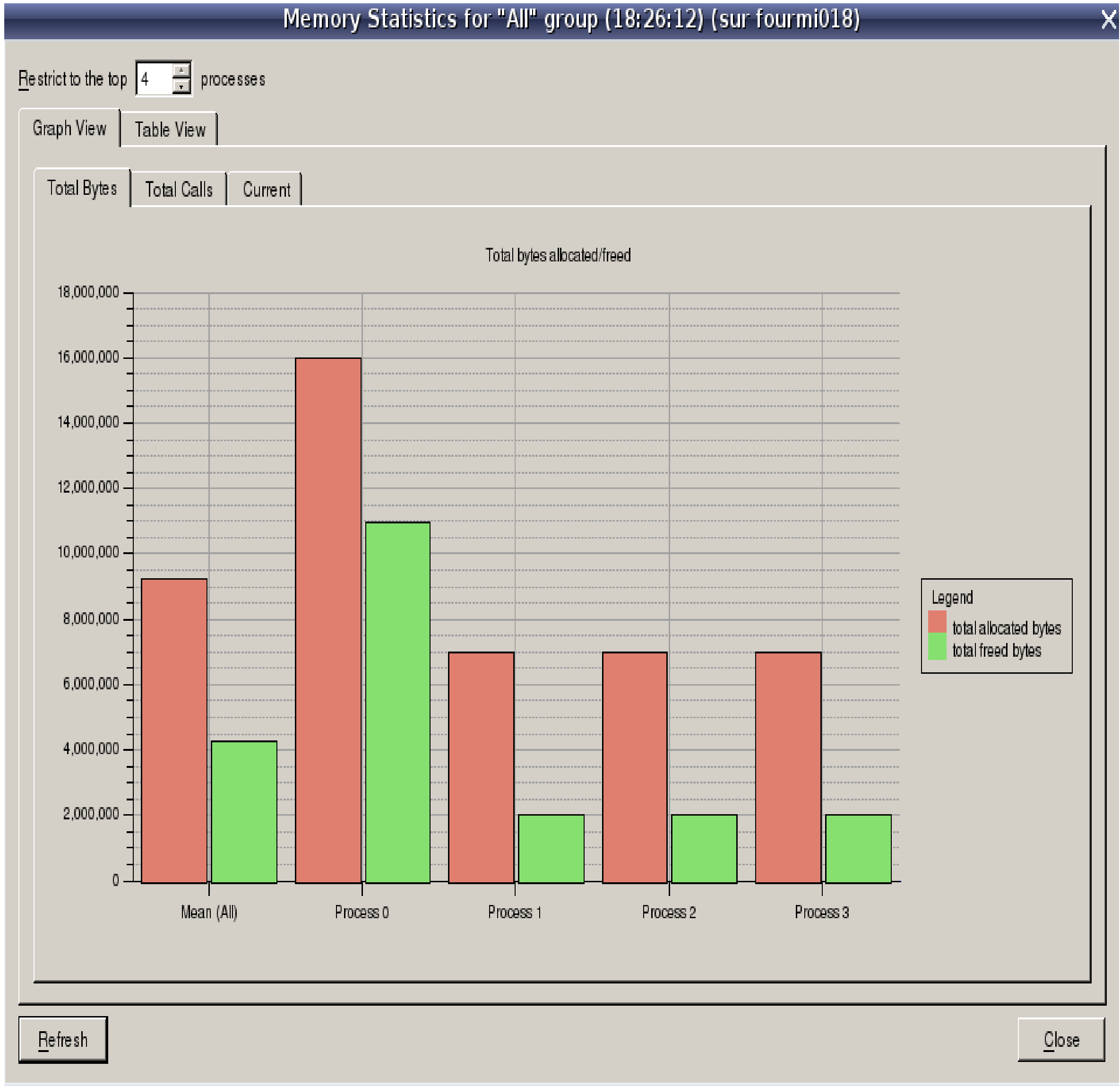
Restrict to the top 4 processes

Show the top 5 locations

Memory Debugging

Memory Statistics

- The Memory Statistics view (View → Overall Memory Stats) shows a total of memory usage across the processes in an application.
- This window displays the total amount of memory allocated/freed since the program began in the left-hand pane. This can help show if your application is unbalanced, if particular processes are allocating or failing to free memory and so on.
- It also shows the total number of calls to allocate/free functions by process. At the end of program execution you can usually expect the total number of calls per process to be similar, and memory allocation calls should always be greater than deallocation calls - anything else indicates serious problems.


Memory Statistics for "All" group (18:26:12) (sur fourmi018)

Restrict to the top 4 processes