TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Matthias Lieber
Center for Information Services and High Performance Computing (ZIH)

Parallel Debugging with DDT

Parallel Programming with MPI, OpenMP, and Tools
Dresden, 8-12 February 2021

Why using a Debugger?

Your program shows incomprehensible behavior, e.g.

— Program terminates abnormally

```
% icc myprog.c -o myprog
% ./myprog
Segmentation fault
```

— Program produces wrong results

```
% ./myprog
Pi = 3.573
```

You want to know what your program is (really) doing

What can a Debugger do?

Observe a running program:

- Print variables (scalars, arrays, structures / derived types, classes)
- Inform about current source code line and function (function call stack)

Control program execution:

- Stop the program at a specific source code line (**Breakpoints**)
- Stop the program when certain conditions are true (**Conditional Breakpoints** and Watchpoints)
- Stop the program before terminating abnormally
- Execute the program line-by-line (**Stepping**)

Typical Usage of a Debugger

Development workflow

- Compile the program with **–g**

```
mpif90 –g myprog.c –o myprog
```

- Run the program under control of the debugger

```
ddt <mpirun command> ./myprog
```

- Use the Debugger to locate the position of the problem and examine variables
- Understand the cause of the problem and correct the source code
- Repeat until problem is solved

Hints:
Always compile your application with the **–g flag**, especially during developing and testing. It adds **symbolic debug info** to the binary and has no performance impact.

Optimizations often interfere with debugging (e.g. functions or variables of interest are “optimized away”). If necessary, compile with the **–O0 flag** to disable optimizations.

Debugger Operation Modes

Start program under debugger control

- Most common way to use a debugger
- Not useful if you want to observe what the program does after a long runtime or you do not expect problems

Attach to an already running program

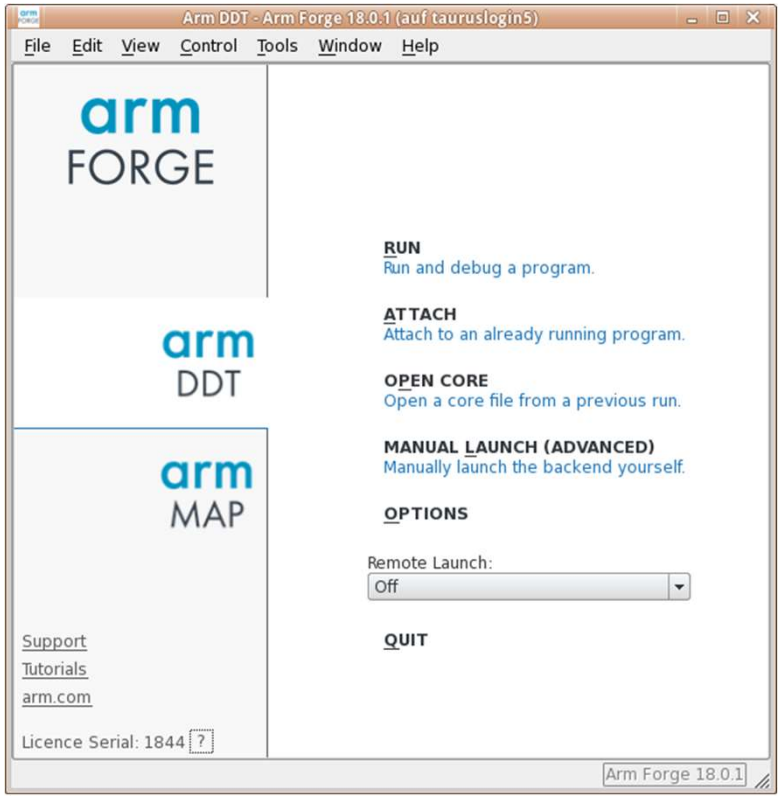
- Program was not started under debugger
- Useful if program has been running for a long time

Core files / core dumps

- Core files are memory state of a crashed program written to file
- Only static analysis of program's data after termination
- Useful if you don't expect a crash or don't want to wait until a crash happens (probably after long runtime)

Arm DDT
Distributed Debugging Tool

- Commercial debugging tool by Arm (Arm acquired Allinea in 2016)
- Languages: C, C++, Fortran
- Parallel Support: Pthreads, OpenMP, MPI, PGAS languages, CUDA, OpenACC
- Available for all common HPC platforms
- Intuitive graphical user interface



- More info: https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/arm-ddt

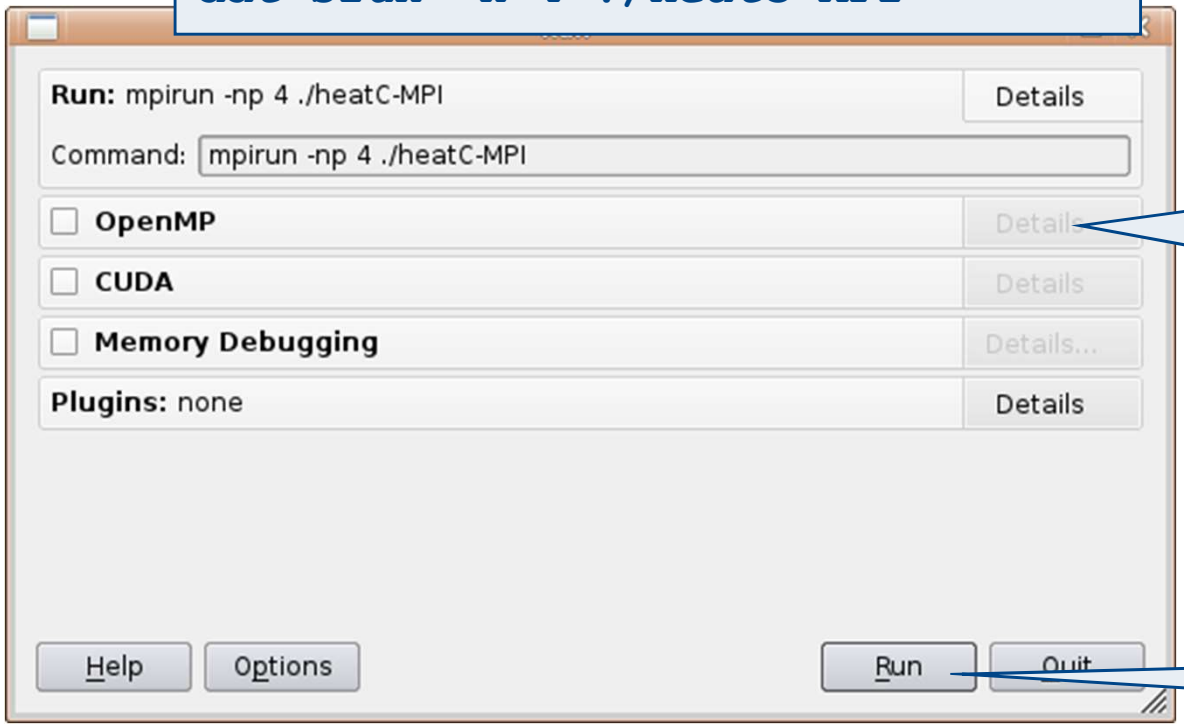
DDT: Program Start

```
% mpicc -g -O0 heatC-MPI.c -o heatC-MPI
% ddt mpirun -np 4 ./heatC-MPI
```

Compile with Debugging

On Taurus we use srun instead of mpirun:
`ddt srun -n 4 ./heatC-MPI`

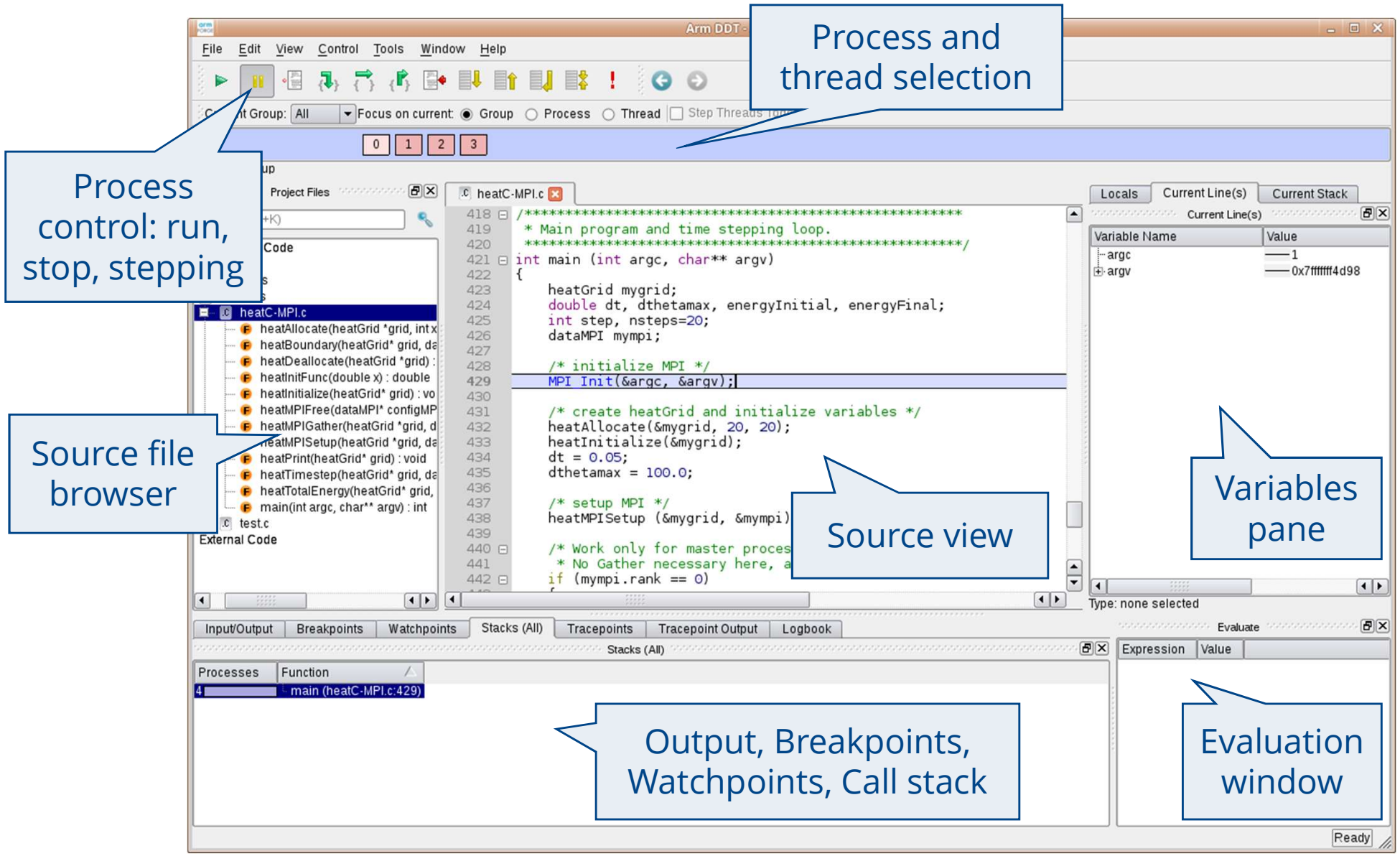
Start DDT: prepend **`ddt`** to mpirun command line



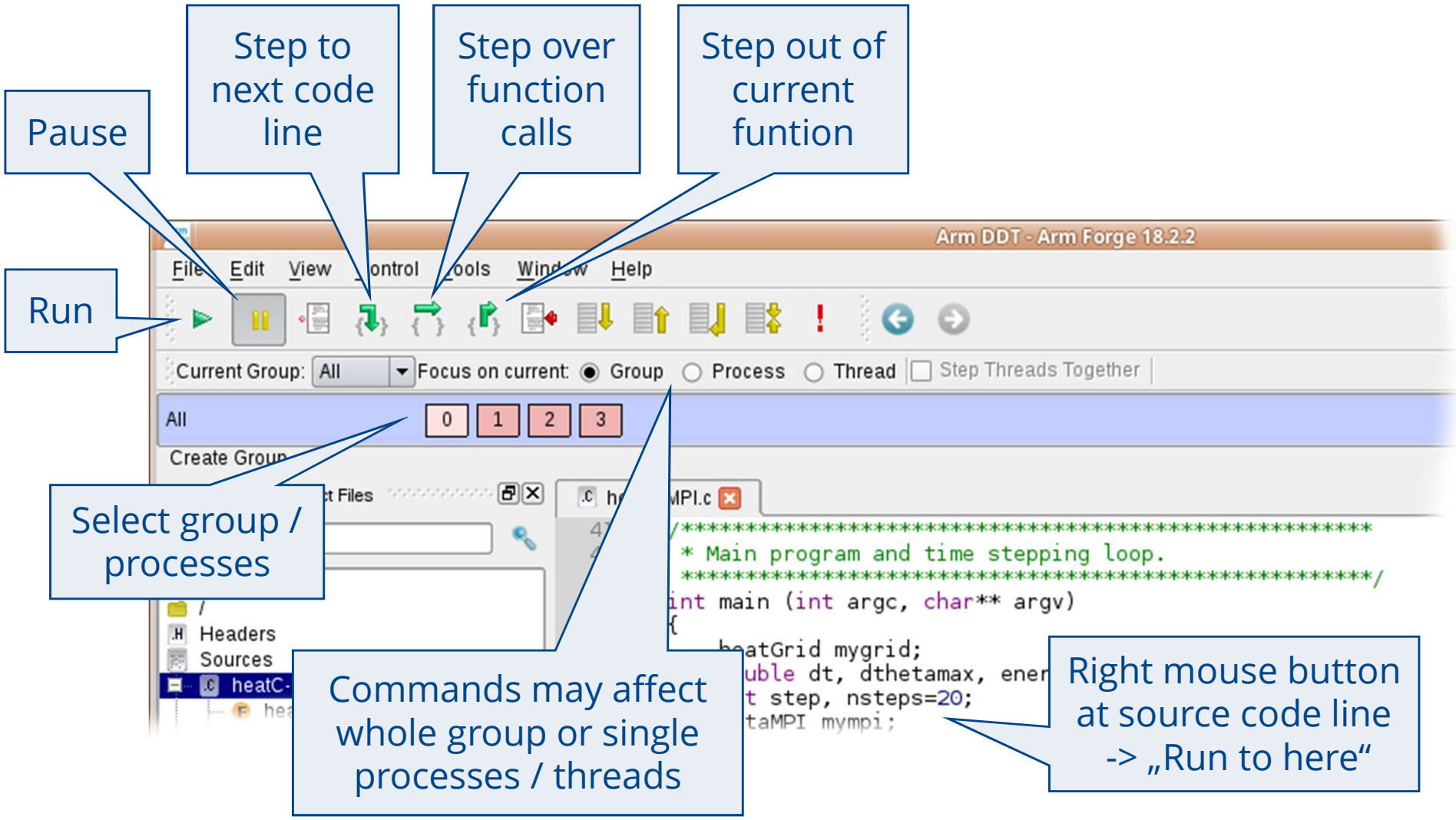
Enable/disbale OpenMP and set number of threads if necessary

Start Program

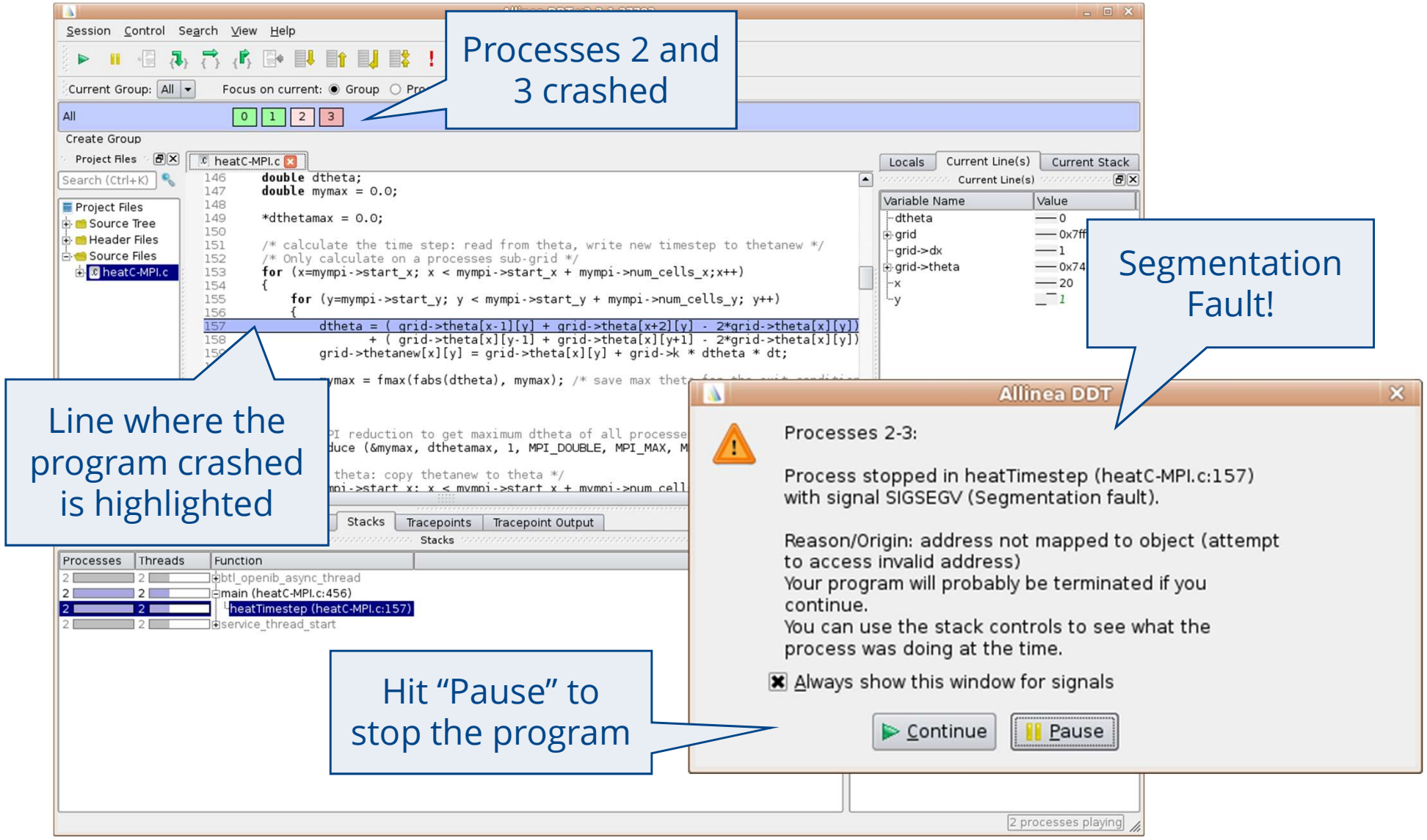
DDT: Main Window



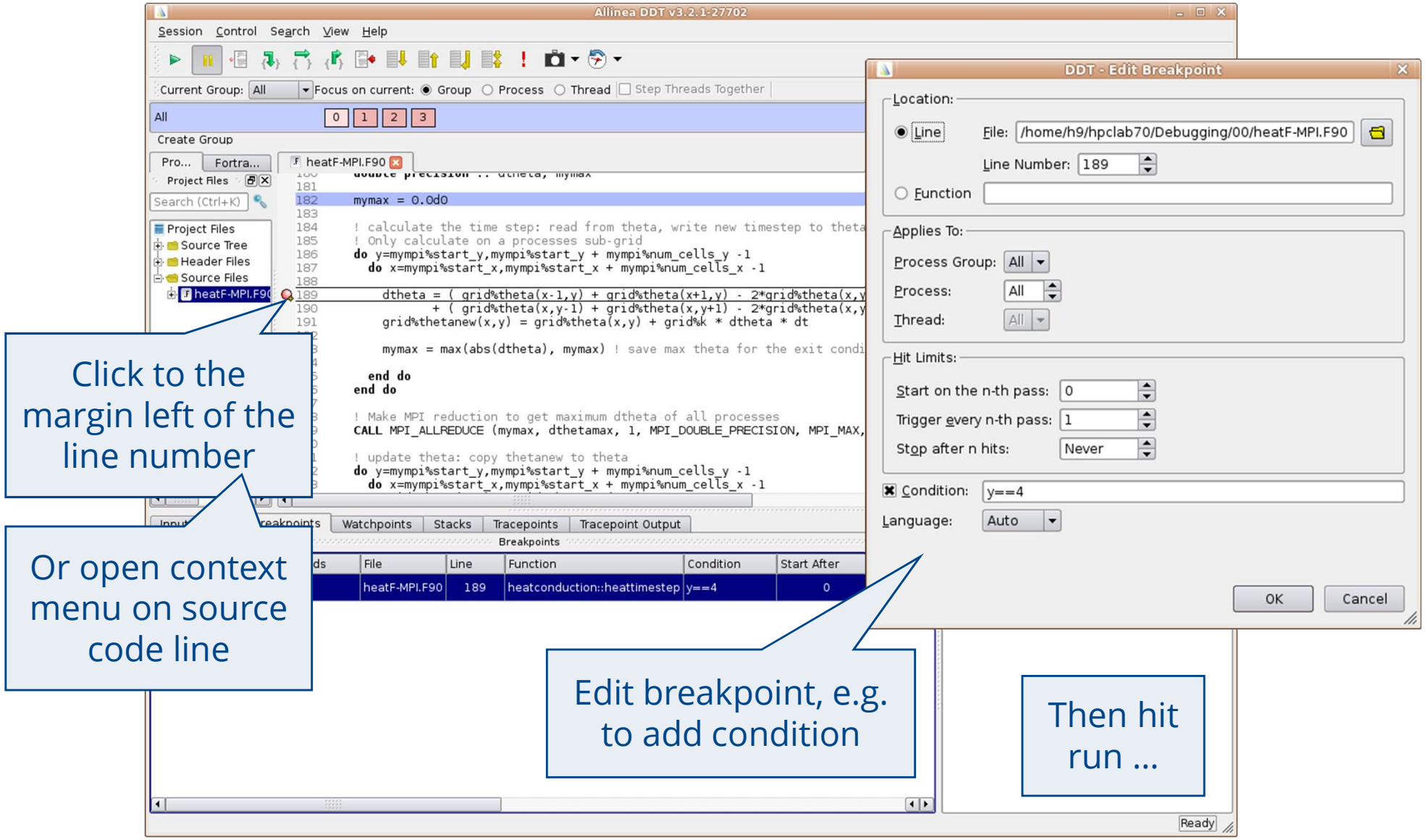
DDT: Process Control & Stepping



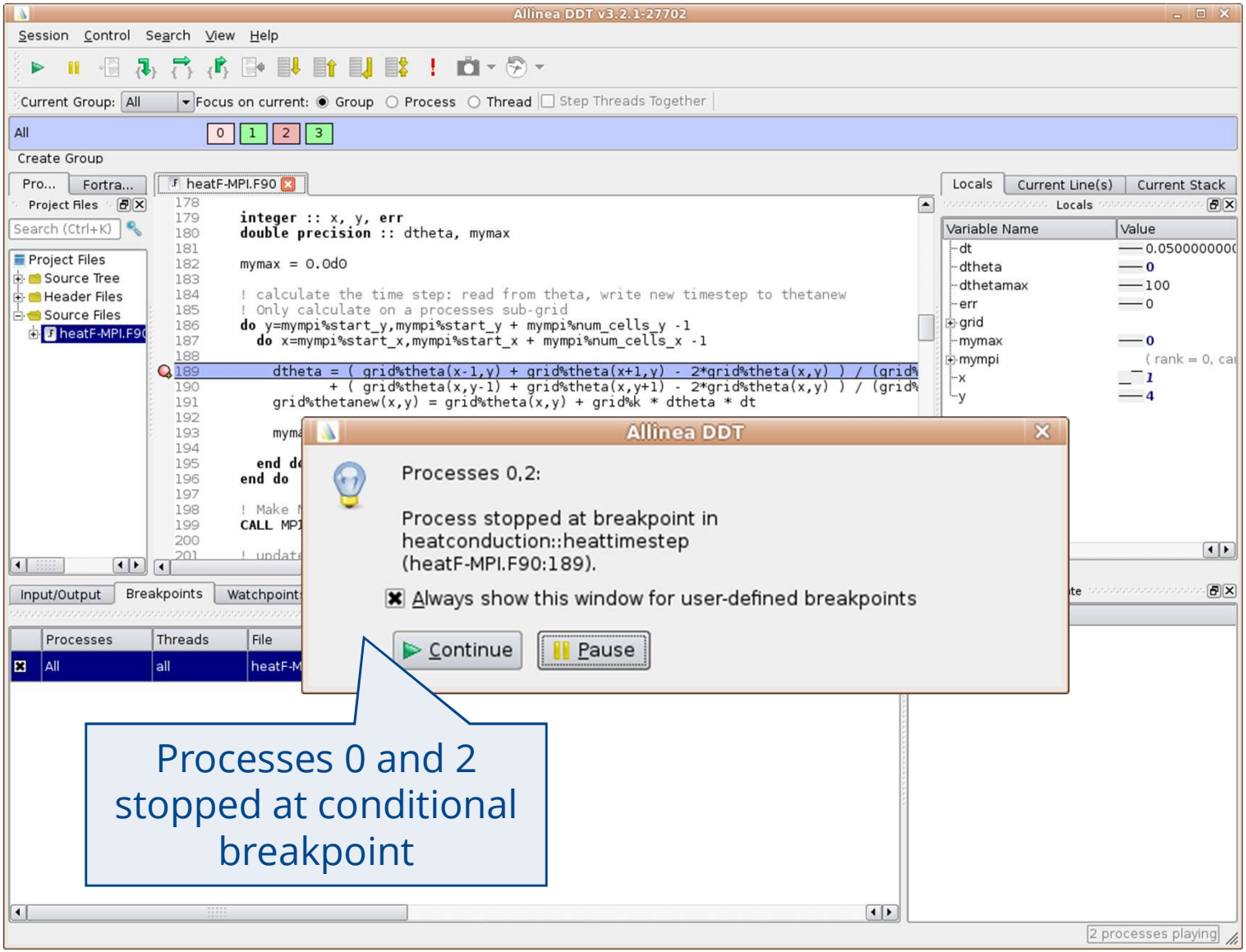
DDT: Segmentation Fault



DDT: Breakpoints (1/2)



DDT: Breakpoints (2/2)



DDT Practical 1: Conditional Breakpoints

C:

```
% cd ~/Debugging/c
% mpicc -g -O0 heatC-MPI.c -o heatC-MPI
% ddt srun -n 4 ./heatC-MPI
```

Fortran 90:

```
% cd ~/Debugging/f90
% mpif90 -g -O0 heatF-MPI.F90 -o heatF-MPI
% ddt srun -n 4 ./heatF-MPI
```

In the DDT run window: uncheck OpenMP, CUDA, Mem. debugging and hit run

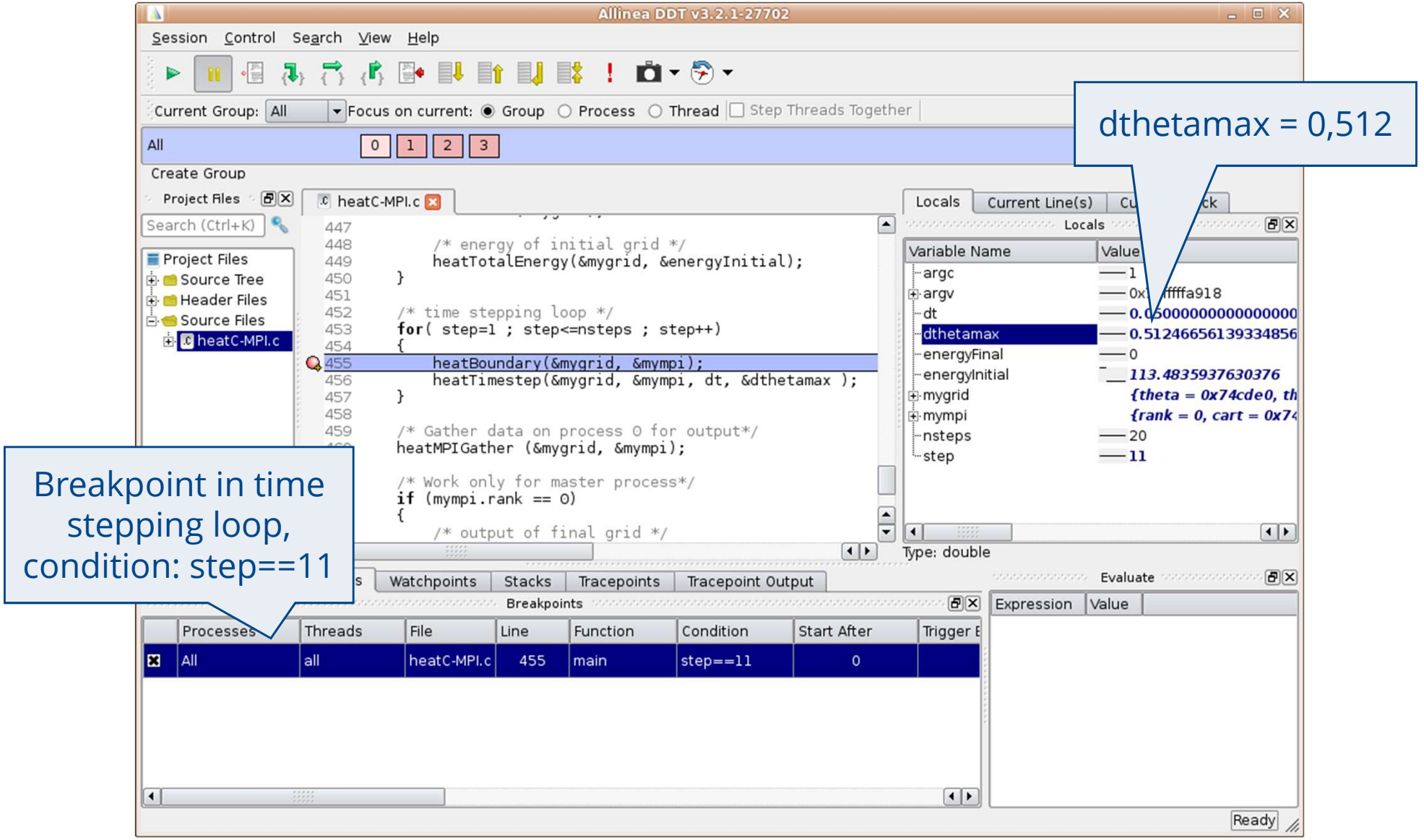
Task A:

- Find out the value of dthetamax after step 10 has been computed.
- Hint: Use a conditional breakpoint in the time stepping loop (main program)

Task B (optional):

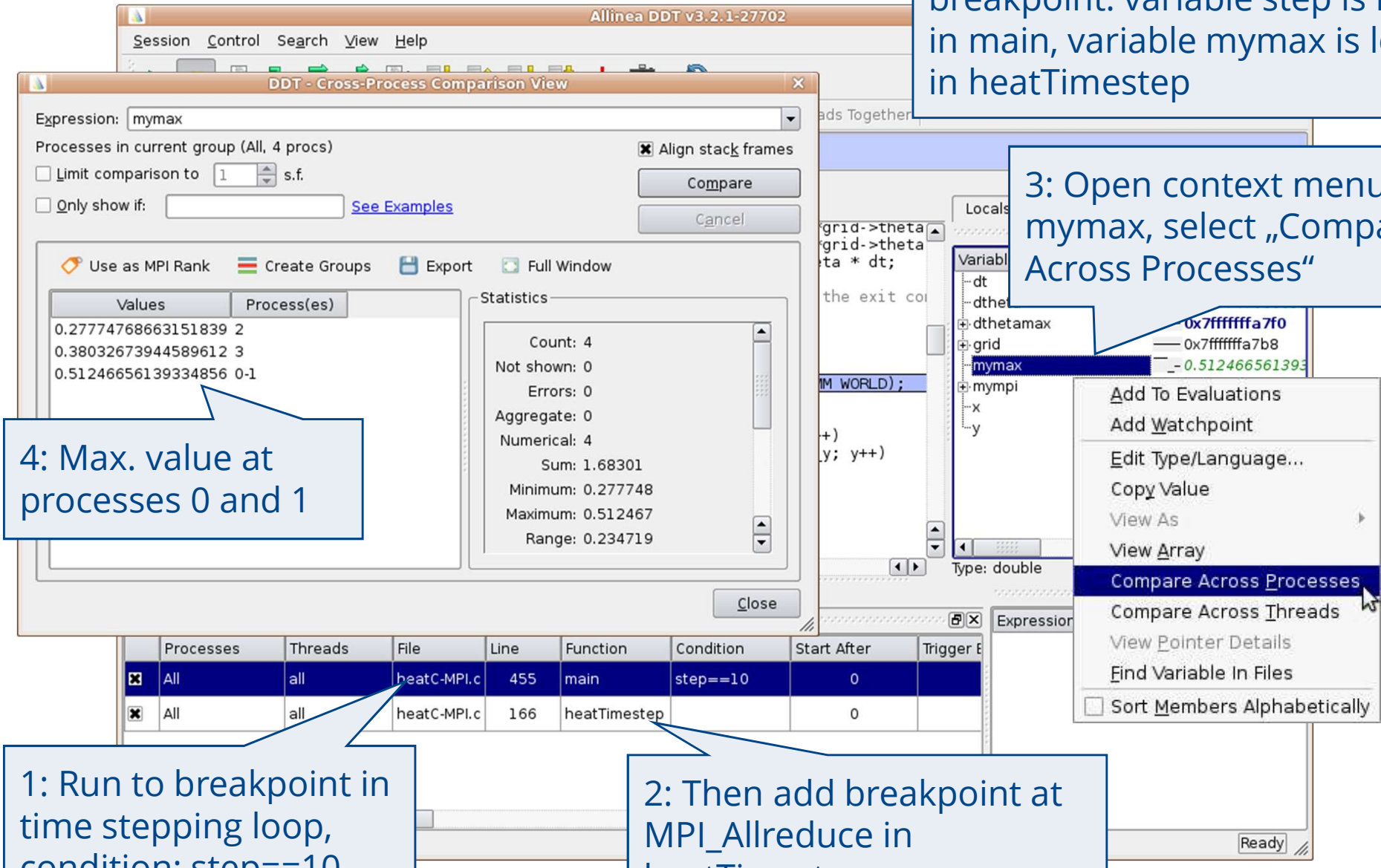
- Which process contributed the maximum to dthetamax at the MPI_Allreduce in heatTimestep after step 10 has been computed?
- Hint: use an additional breakpoint at the MPI_Allreduce, then right click on the variable mymax in the variables pane and select “Compare Across Processes”

DDT Practical 1: Task A Solution



DDT Practical 1: Task B Solution

Not possible with a single breakpoint: variable step is local in main, variable mymax is local in heatTimestep



3: Open context menu for mymax, select „Compare Across Processes“

4: Max. value at processes 0 and 1

1: Run to breakpoint in time stepping loop, condition: step==10

2: Then add breakpoint at MPI_Allreduce in heatTimestep

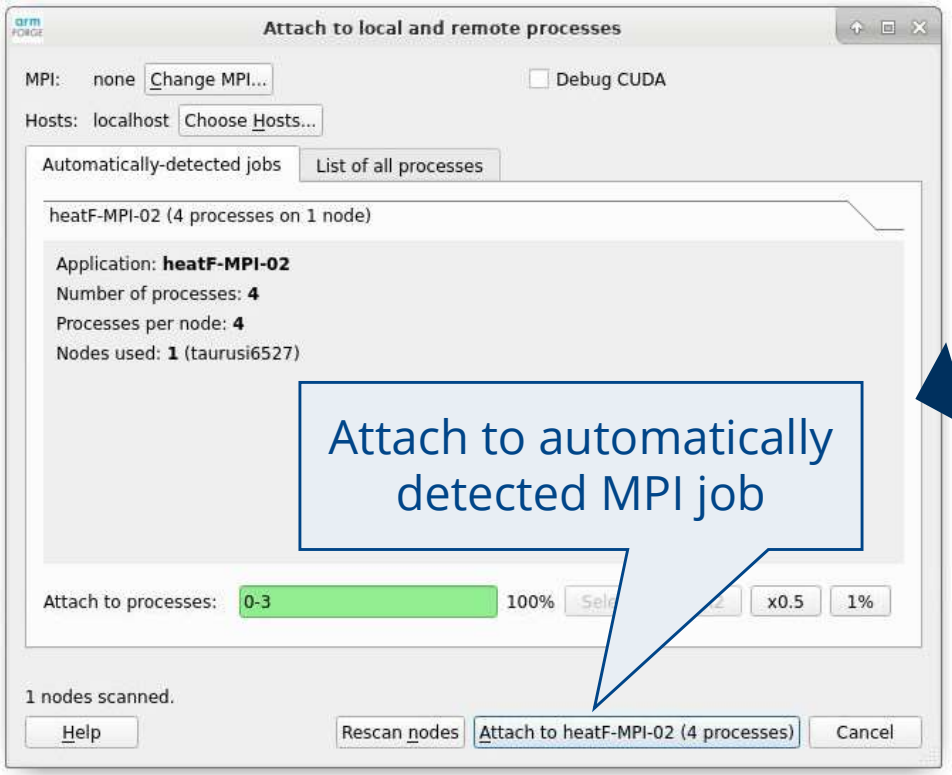
DDT: Attach to running program

```
% mpif90 -g heatF-MPI-02.F90 -o heatF-MPI-02
% srun -n 4 ./heatF-MPI-02
. . .
```

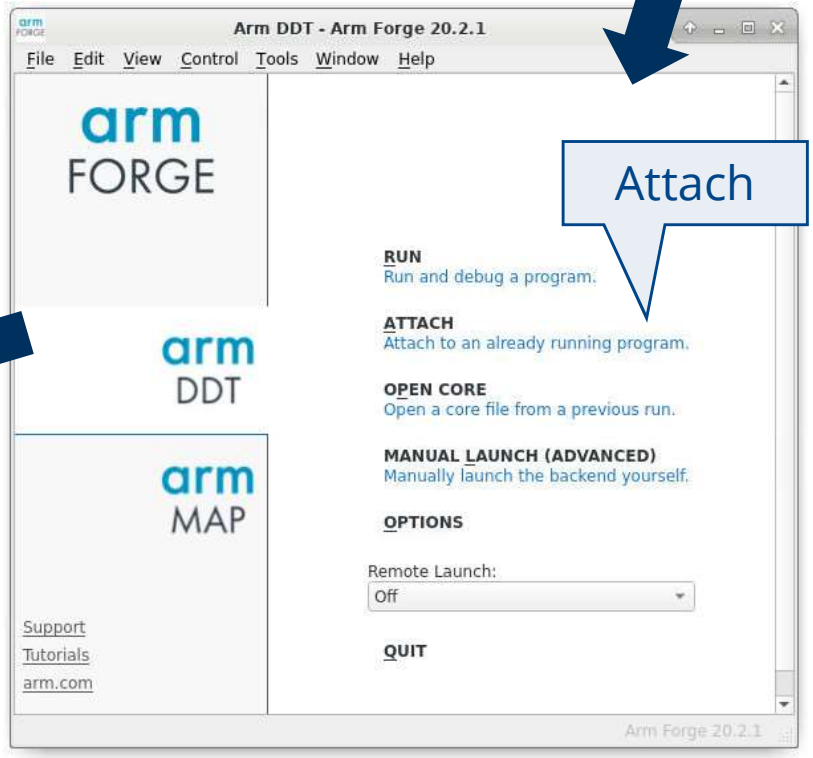
Program runs – you want to know what it is doing?

Start DDT in a 2nd terminal

```
% ddt
```



Attach to automatically detected MPI job



Attach

DDT: Core Files (1/2)

```
% mpif90 -g -O0 heatF-MPI-01.F90 -o heatF-MPI-01
% ulimit -c
0
% ulimit -Sc 100000
% export FOR_DUMP_CORE_FILE=yes
% srun -n 2 ./heatF-MPI-01

. . .

forrtl: severe (174): SIGSEGV, segmentation fault occurred

. . .

srun: error: taurusi6595: tasks 0-1: Exited with exit code 174

. . .

% ls -lh core*
-rw------- 1 gpu59 1111111 42M Jan 27 13:21 core.taurusi6595.taurus.hrsk.tu-dresden.de.3371
-rw------- 1 gpu59 1111111 42M Jan 27 13:21 core.taurusi6595.taurus.hrsk.tu-dresden.de.3372
% ddt
```

Check core file size limit (reports kB) and increase if required (sets to 100 MB)

Intel Fortran only

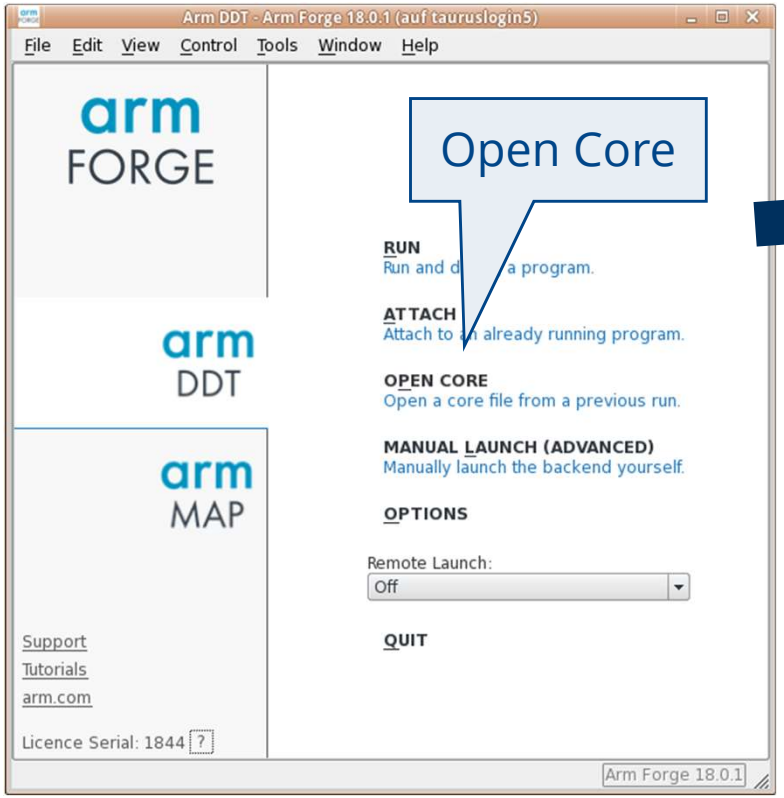
Segmentation Fault

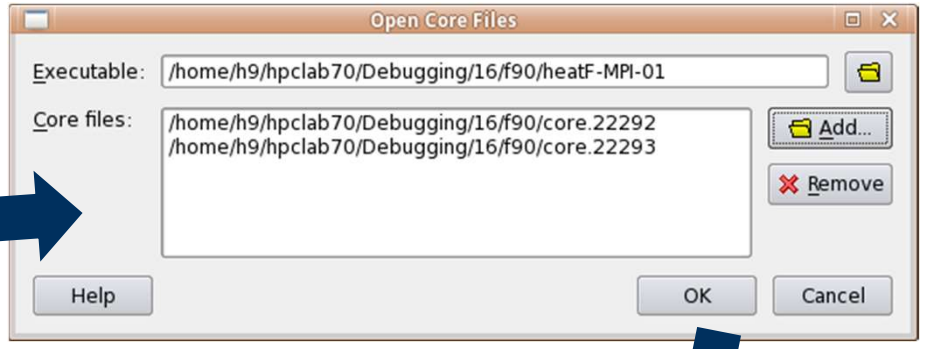
srun realizes crash

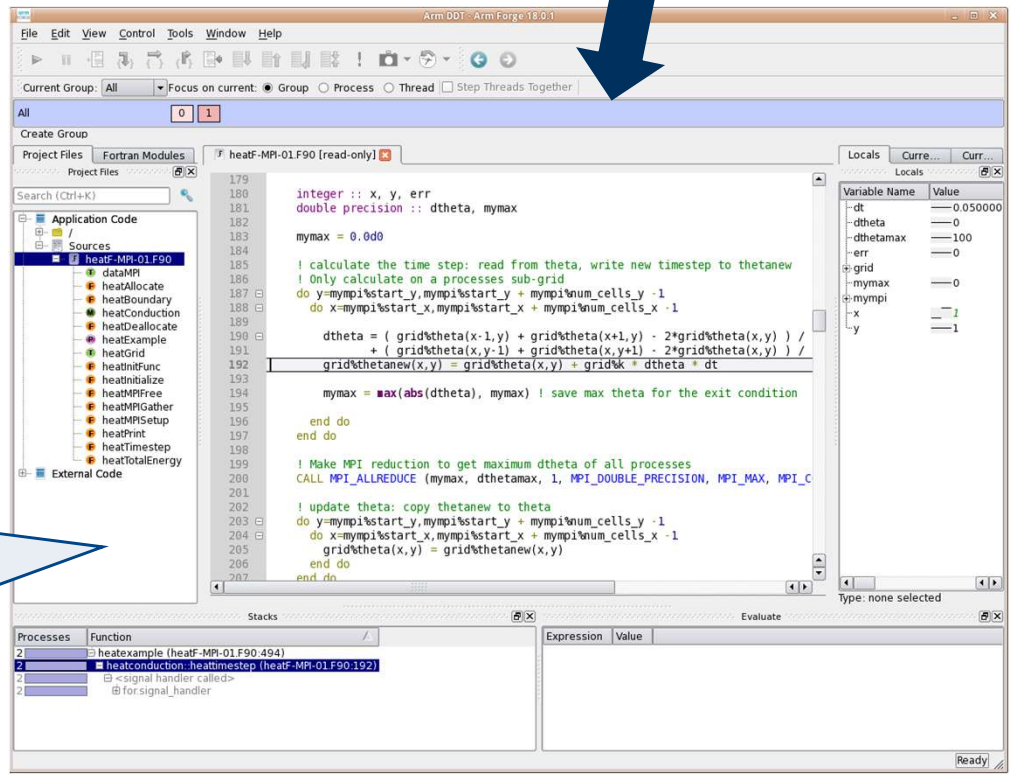
Per-process core files

Analyze with DDT

DDT: Core Files (2/2)



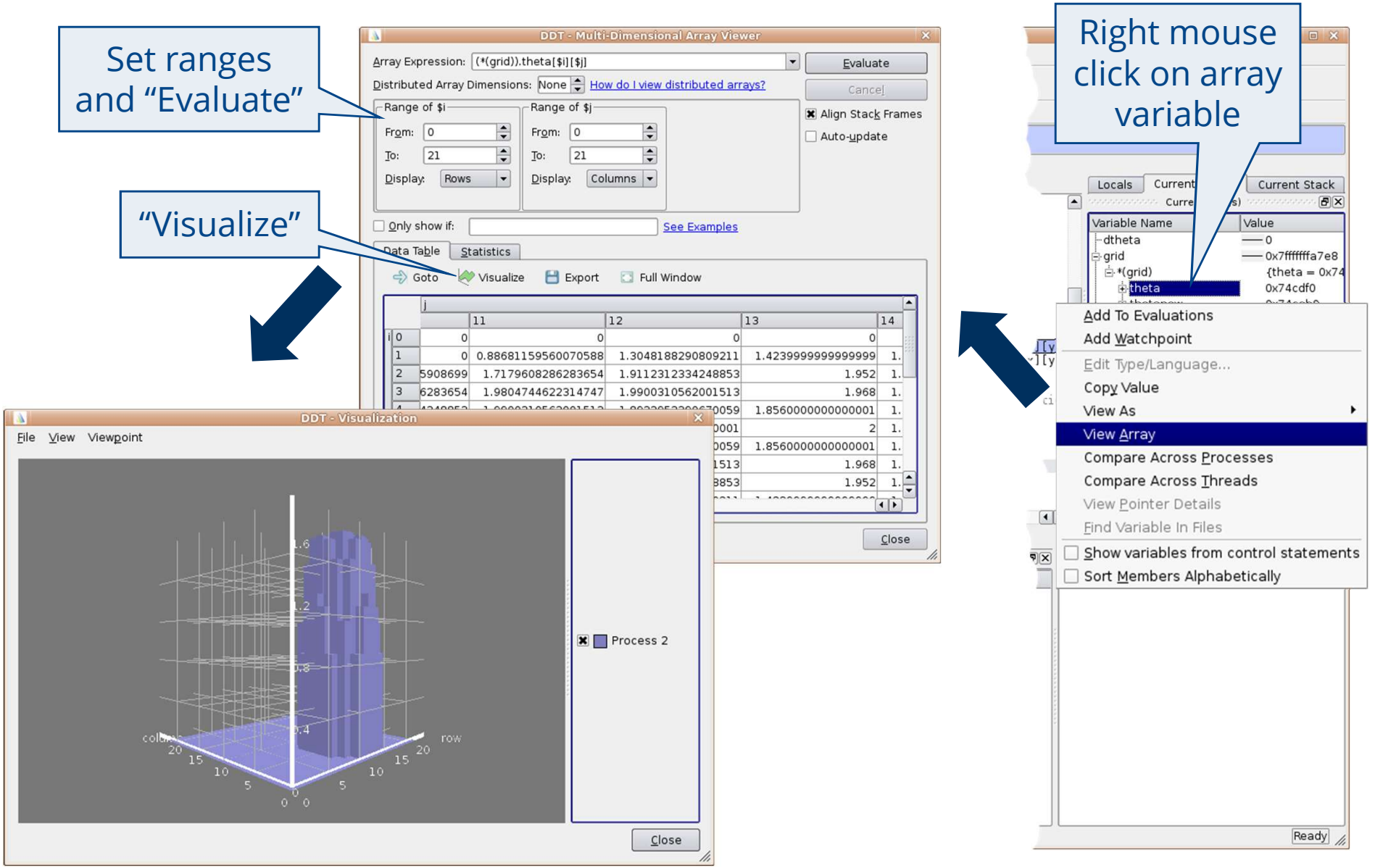




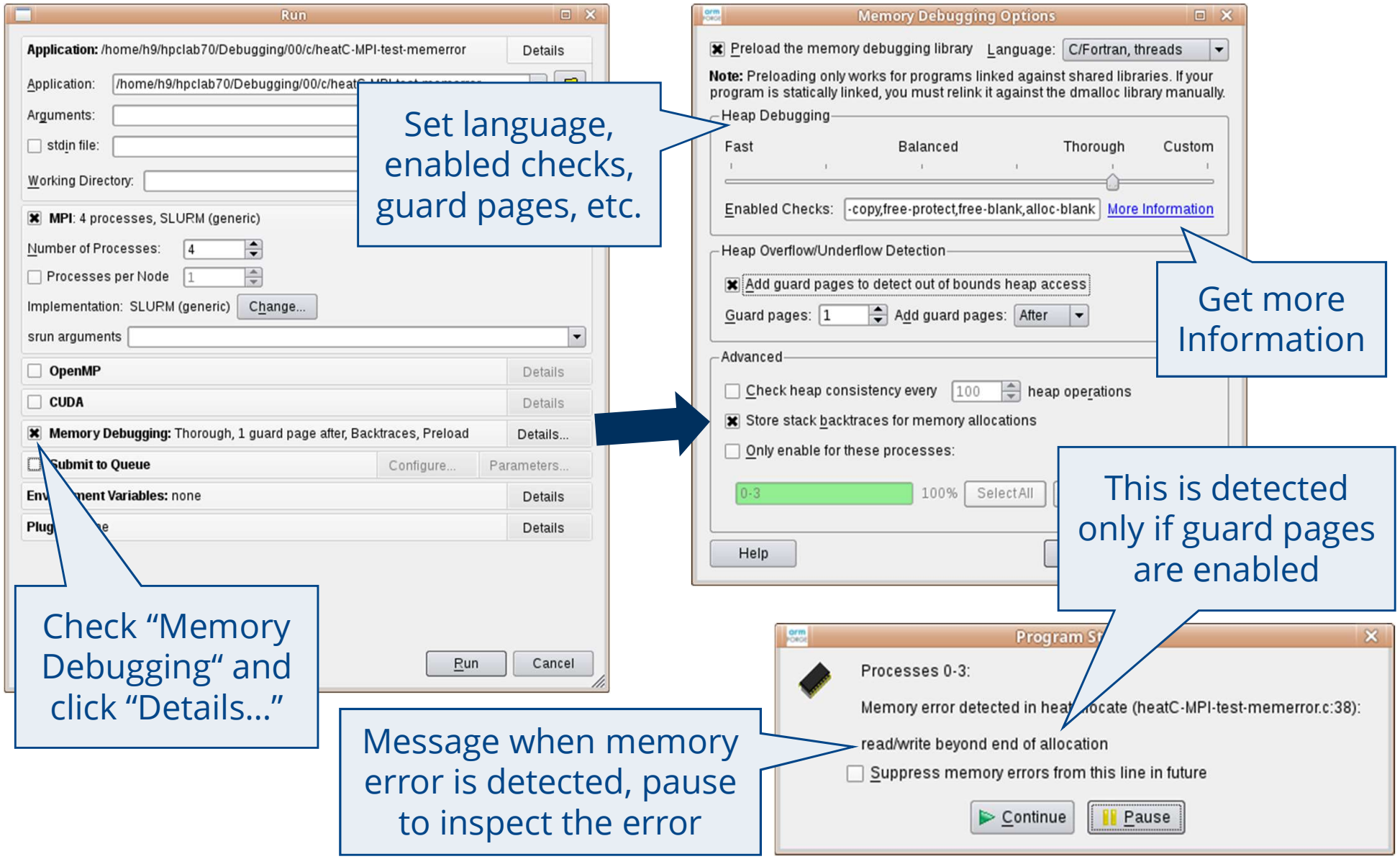
DDT shows position of the crash in the source code and values of variables at the time of the crash.

But no running or stepping possible!

DDT: Multidimensional Array Visualization



DDT: Memory Debugging



DDT Practical 2: Find the Bugs!

Find the bug in each of the three programs!

- Compile and first run normally (4 processes) to observe the behavior, then use DDT to find the bug
- If the program stops in MPI, DDT may complain about missing source files: ignore and select an application function in the call stack view

`heatC-MPI-01 / heatF-MPI-01`

- Produce core dumps (with up to 4 MPI processes) and open with DDT
- You already know this example from the Intro's practical

`heatC-MPI-02 / heatF-MPI-02`

- Run without DDT and then attach DDT (use a second terminal session)
- In case of trouble when attaching: ensure that Options - System – Debugger is set to GNU 7.6.2

`heatC-MPI-03 / heatF-MPI-03`

- Hint: compare arguments of send and receive call

Optional OpenMP task on next slide

DDT Practical 3: Breakpoints with OpenMP (optional)

C:

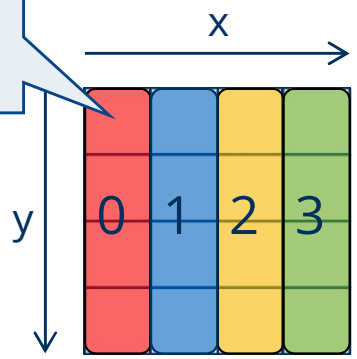
```
% cd ~/Debugging/c
% icc -g -O0 -fopenmp heatC-omp.c -o heatC-omp
% ddt ./heatC-omp
```

Fortran 90:

```
% cd ~/Debugging/f90
% ifort -g -O0 -fopenmp heatF-omp.F90 -o heatF-omp
% ddt ./heatF-omp
```

Task:

— Run with 4 threads in DDT and find out which thread computes which part of the 20 x 20 grid



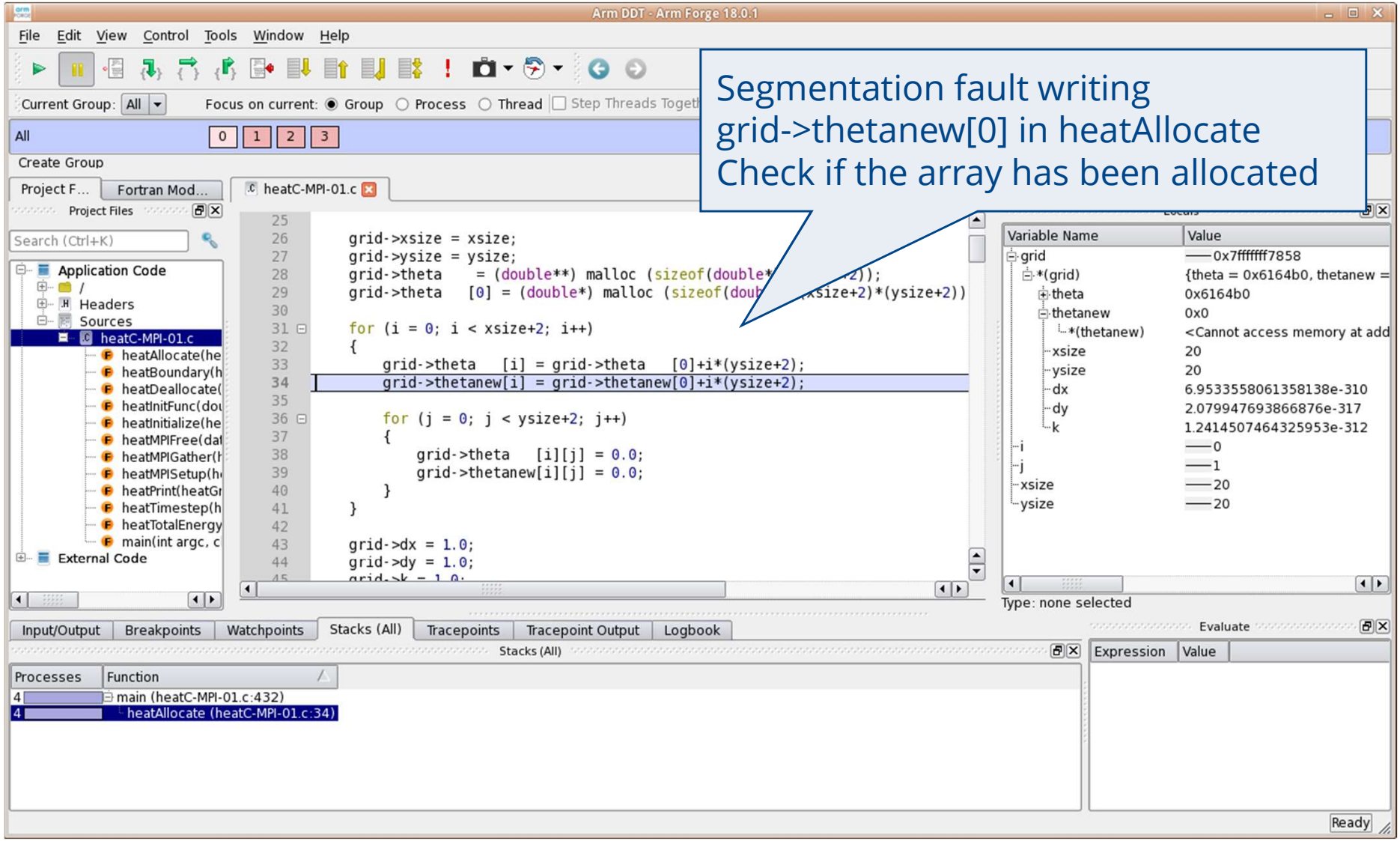
Hints:

— Use a breakpoint in the inner compute loop in heatTimestep and examine loop variable x, use condition y==1 to jump to the next iteration of the x-loop

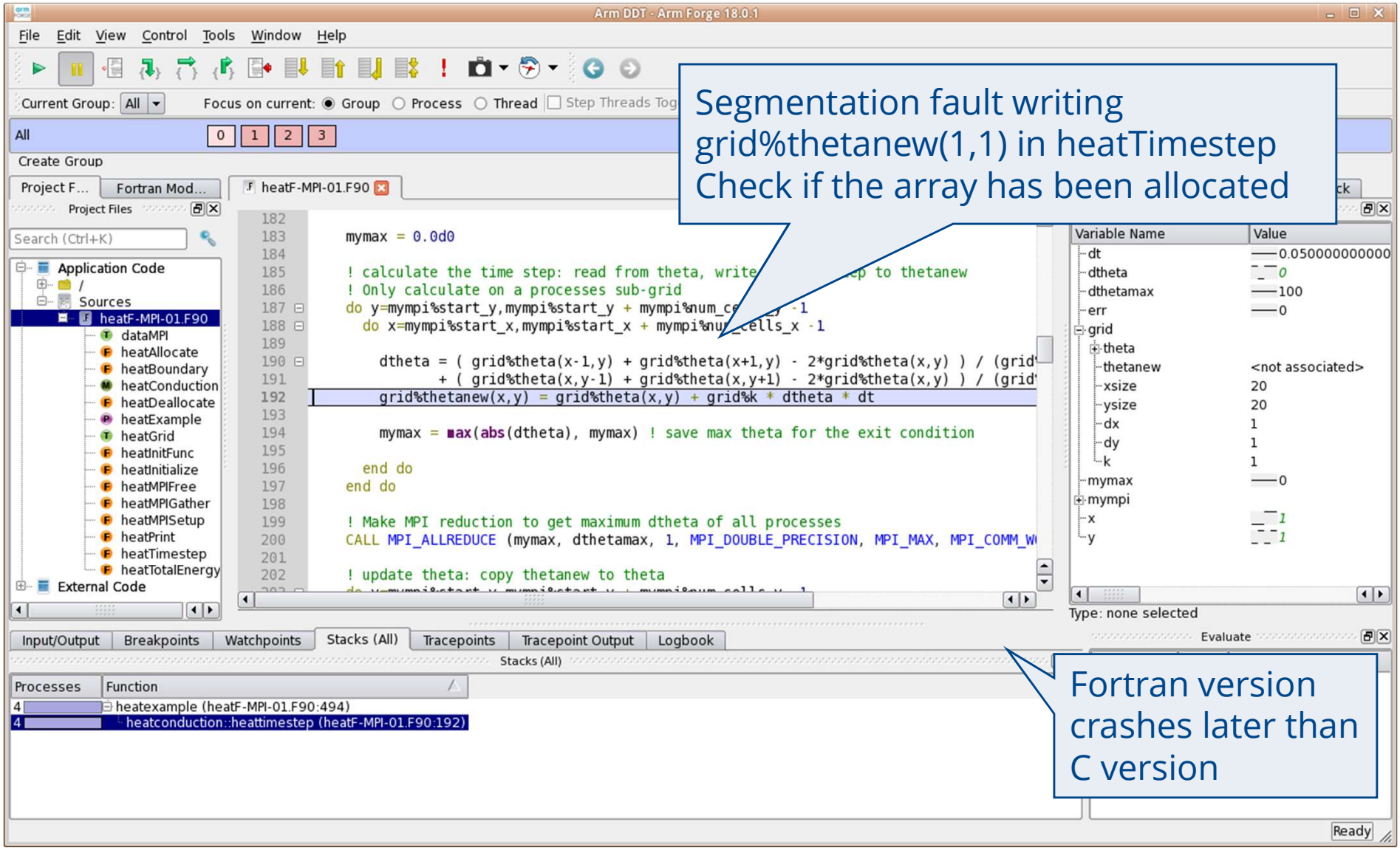
— Fortran: x and y are interchanged compared to C, y-loop is parallelized: need to examine y

— Breakpoints in OpenMP parallel regions sometimes behave unexpected - it helps to run a single thread, while pausing all others: select “Focus on current thread” and select the thread below

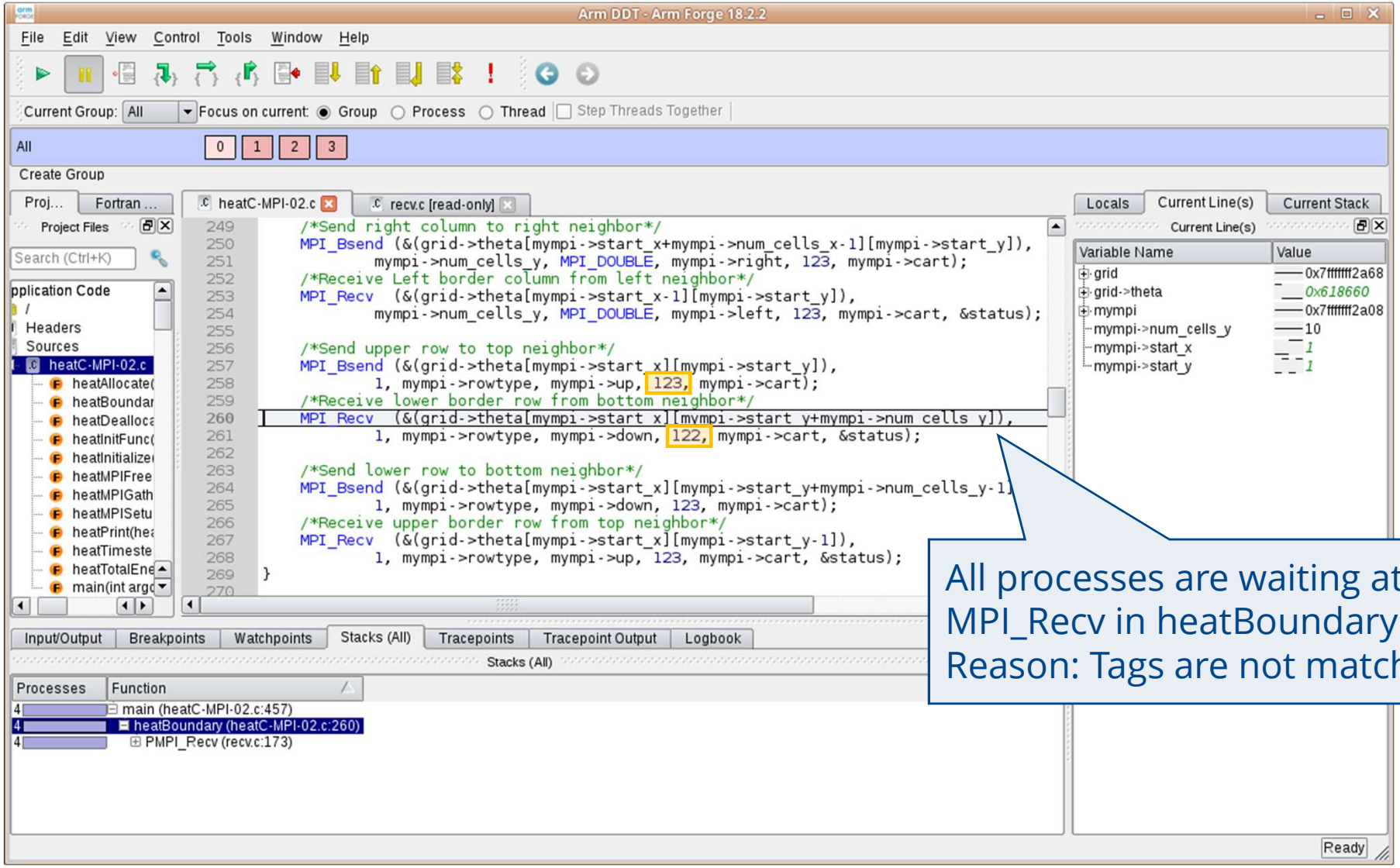
DDT Practical 2 Task 01: Solution (C)



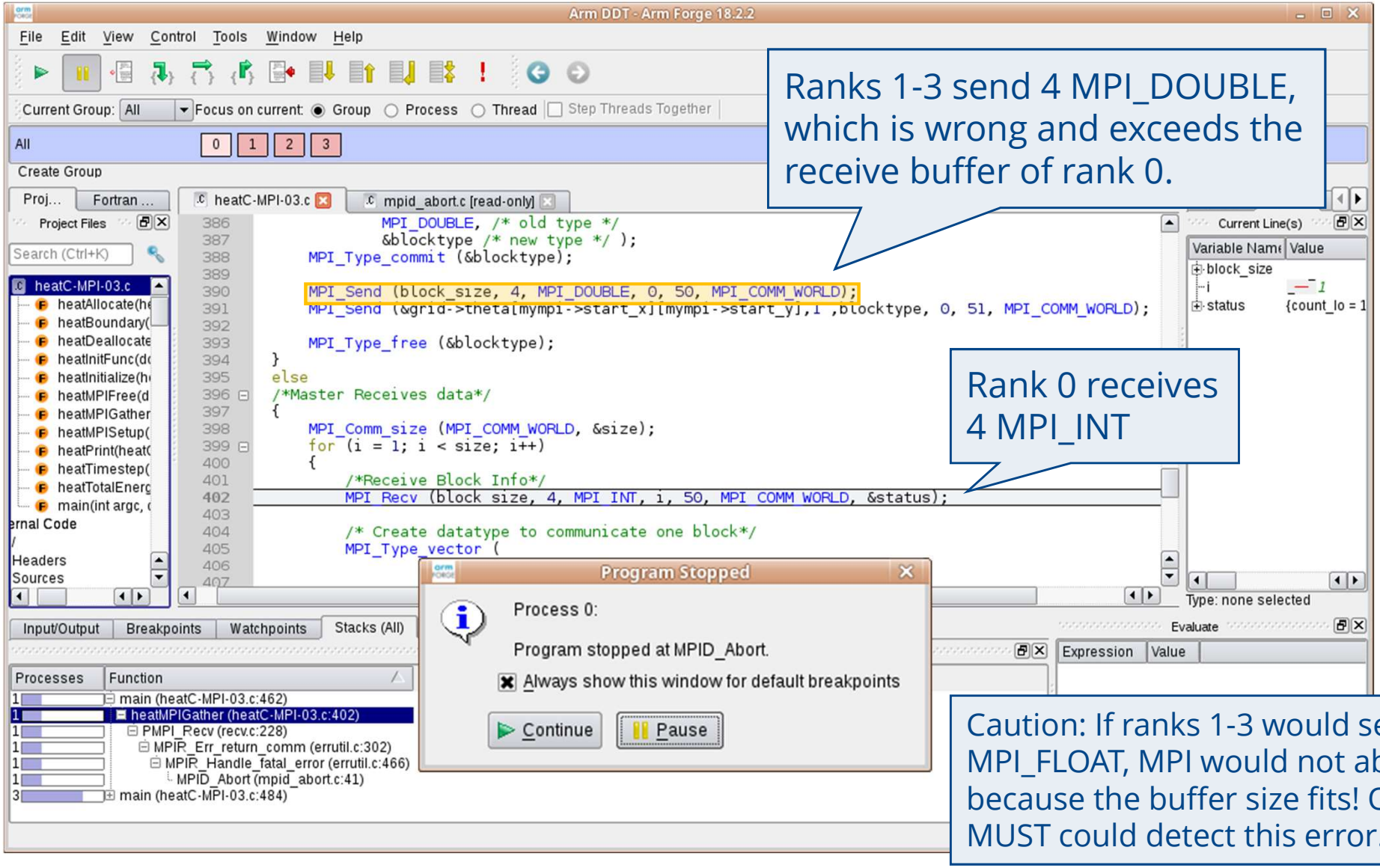
DDT Practical 2 Task 01: Solution (Fortran)



DDT Practical 2 Task 02: Solution



DDT Practical 2 Task 03: Solution



DDT Practical 3: Solution