Debugging with GDB and DDT

Ramses van Zon
SciNet HPC Consortium
University of Toronto

May 10, 2013

Outline

- Debugging Basics
- Debugging with the command line: GDB
- Debugging with DDT

Debugging basics

Debugging basics

Help, my program doesn't work!

```
$ gcc -O3 answer.c
$ ./a.out
Segmentation fault
```

↓

a miracle occurs

↓

My program works brilliantly!

```
$ gcc -O3 answer.c
$ ./a.out
42
```

- Unfortunately, "miracles" are not yet supported by SciNet.

Debugging:

Methodical process of finding and fixing flaws in software

Common symptoms

Errors at compile time

- Syntax errors: easy to fix
- Library issues
- Cross-compiling
- Compiler warnings
  Always switch this on, and fix or understand them!

But just because it compiles does not mean it is correct!

Runtime errors

- Floating point exceptions
- Segmentation fault
- Aborted
- Incorrect output (nans)

Common issues

Arithmetic	corner cases (`sqrt(-0.0)`), infinities
Memory access	Index out of range, uninitialized pointers.
Logic	Infinite loop, corner cases
Misuse	wrong input, ignored error, no initialization
Syntax	wrong operators/arguments
Resource starvation	memory leak, quota overflow
Parallel	race conditions, deadlock

What is going on?

- Almost always, a condition you are sure is satisfied, is not.
- But your programs likely relies on many such assumptions.
- First order of business is finding out what goes wrong, and what assumption is not warranted.
- *Debugger:* program to help detect errors in other programs.
- **You are the real debugger.**

Ways to debug

- Preemptive:
  - Turn on compiler warnings: fix or understand them!

    $ `gcc/gfortran -Wall`
  - Check your assumptions (e.g. use **`assert`**).
- Inspect the exit code and read the error messages!
- Use a debugger
- Add print statements ←**No way to debug!**

What's wrong with using print statements?

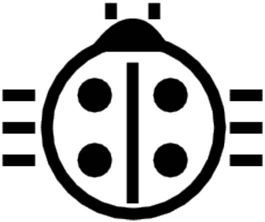
Strategy

- Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output *bug not found?*
- Removing the extra code after the bug is fixed
- Repeat for each bug

Problems with this approach

- Time consuming
- Error prone
- Changes memory, timing. . . **There's a better way!**

Symbolic debuggers



Symbolic debuggers

Features

1. Crash inspection
2. Function call stack
3. Step through code
4. Automated interruption
5. Variable checking and setting

Use a graphical debugger or not?

- Local work station: graphical is convenient
- Remotely (SciNet): can be slow

In any case, graphical and text-based debuggers use the same concepts.

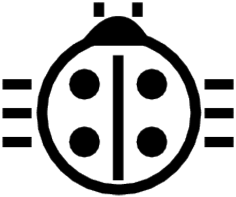
Symbolic debuggers

Preparing the executable

- Add required compilination flags:
  `$ gcc/g++/gfortran -g -gstabs`
  `$ icc/icpc/ifort -g -debug parallel`
  `$ nvcc -g -G`
- Optional: switch off optimization **-O0**

Command-line based symbolic debuggers: gdb

GDB



What is GDB?

- Free, GNU license, symbolic debugger.
- Available on many systems.
- Been around for a while, but still developed and up-to-date
- Text based, but has a '-tui' option.

```
$ module load gcc
$ gcc -g -O0 example.c -o example
$ module load gdb
$ gdb -tui example
...
(gdb)_
```

GDB basic building blocks

GDB building block #1: Inspect crashes

Inspecting core files

Core = file containing state of program after a crash

- needs max core size set (**`ulimit -c <number>`**)
- gdb reads with **`gdb <executable> <corefile>`**
- it will show you where the program crashed

No core file?

- can start gdb as **`gdb <executable>`**
- type **`run`** to start program
- gdb will show you where the program crashed if it does.

SciNet compute • calcul CANADA

GDB building block #2: Function call stack

Interrupting program

- Press Crtl-C while program is running in gdb
- gdb will show you where the program was.

Stack trace

- From what functions was this line reached?
- What were the arguments of those function calls?

gdb commands

backtrace	function call stack
continue	continue
down	go to called function
up	go to caller

GDB building block #3: Step through code

Stepping through code

- Line-by-line
- Choose to step into or over functions
- Can show surrounding lines or use **`-tui`**

gdb commands

`list`	list part of code
`next`	continue until next line
`step`	step into function
`finish`	continue until function end
`until`	continue until line/function

GDB building block #4: Automatic interruption

Breakpoints

- **`break [file:]<line>|<function>`**
- each breakpoint gets a number
- when run, automatically stops there
- can add conditions, temporarily remote breaks, etc.

Related gdb commands

`delete`	unset breakpoint
`condition`	break if condition met
`disable`	disable breakpoint
`enable`	enable breakpoint
`info breakpoints`	list breakpoints
`tbreak`	temporary breakpoint

GDB building block #5: Variables

Checking a variable

- Can print the value of a variable
- Can keep track of variable (print at prompt)
- Can stop the program when variable changes
- Can change a variable ("what if ...")

gdb commands

`print`	print variable
`display`	print at every prompt
`set variable`	change variable
`watch`	stop if variable changes

Demonstration GDB

```
$ ssh USER@login.scinet.utoronto.ca -X
$ ssh gpc01 -X
$ qsub -l nodes=1:ppn=8,walltime=4:00:00 -I -X
$ cd $SCRATCH
$ cp -r /scinet/course/ss2013 .
$ cd ss2013/HPC106_debug/code
$ source setup
$ cd ex1
$ make dbgtest #(or dbgtestf)
$ ulimit -c 1024
$ ./dbgtest #(or dbgtestf)
Hello
Hi
```

You'll find that the latter does not work. Start up

```
$ gdb -tui dbgtest #(or dbgtestf)
```

Graphical symbolic debuggers

Graphical symbolic debuggers

Features

- Nice, more intuitive graphical user interface
- Front to command-line based tools: Same concepts
- Need graphics support: X forwarding (or VNC)

Available on SciNet: ddd and ddt

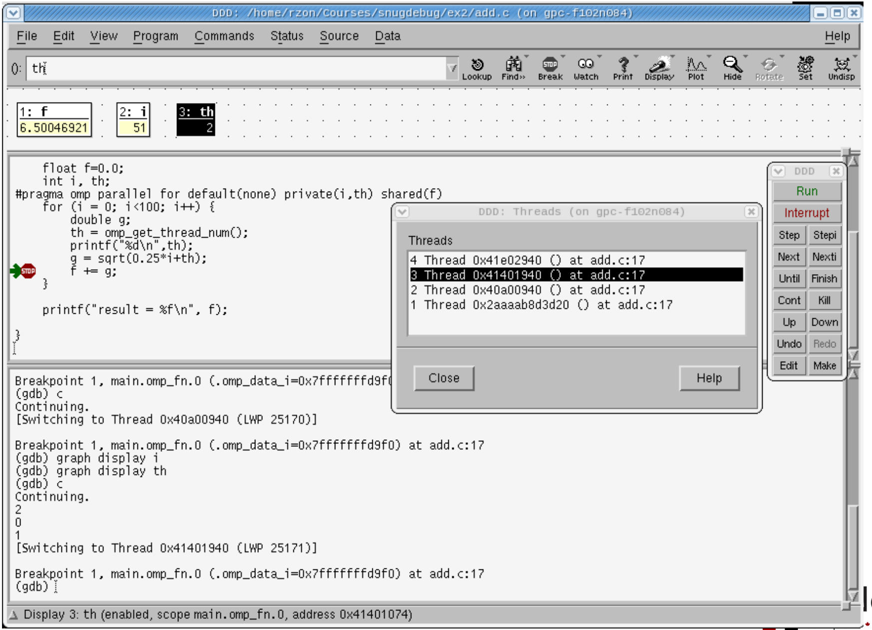
- ddd

  ```
  $ module load gcc ddd
  $ ddd <executable compiled with -g flag>
  ```

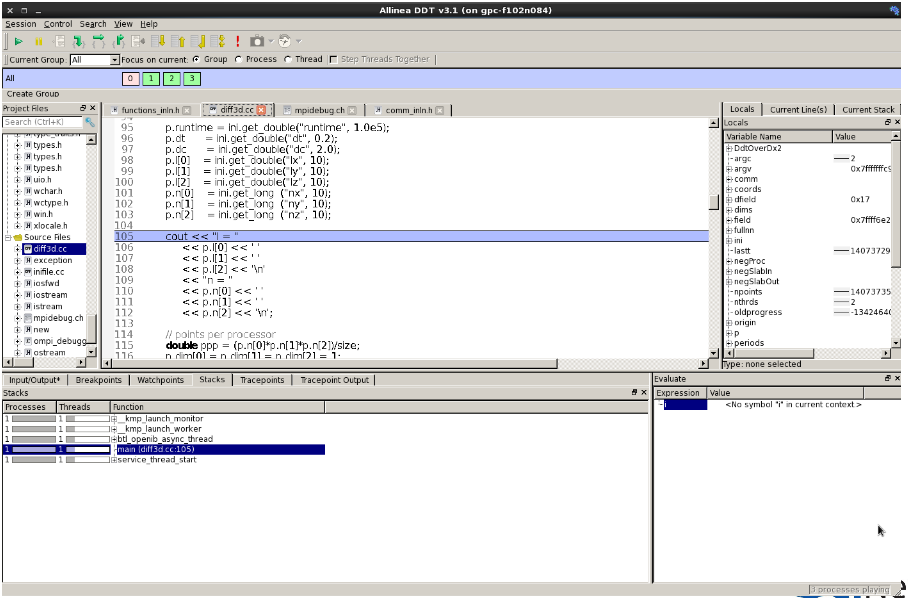
- ddt

  ```
  $ module load ddt
  $ ddt <executable compiled with -g flag>
  (more later)
  ```

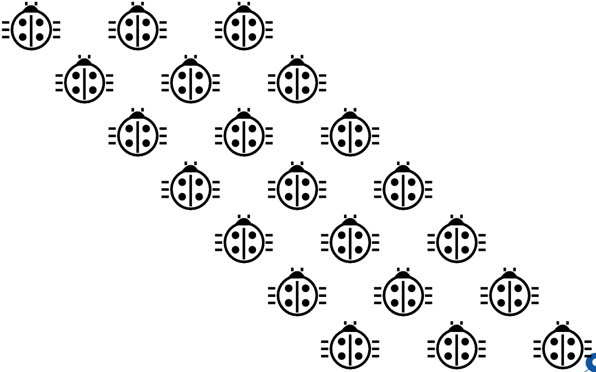
Graphical symbolic debuggers - ddd



Graphical symbolic debuggers - ddt



Parallel debugging



Parallel debugging - 1 Shared memory

Use gdb for

- Tracking each thread's execution and variables
- OpenMP serialization: **`p omp_set_num_threads(1)`**
- Stepping into OpenMP block: **`break`** at first line!
- Thread-specific breakpoint: **`b <line> thread <n>`**

Use helgrind for

- Finding race conditions:

```
$ module load valgrind
$ valgrind --tool=helgrind <exe> &> out
$ grep <source> out
```

where **`<source>`** is the name of the source file where you suspect race conditions (valgrind reports a lot more)

Parallel debugging - 2 Distributed memory

Multiple MPI processes

- Your code is running on different cores!
- Where to run debugger?
- Where to send debugger output?
- Much going on at same time.
- No universal free solution.

Good approach:

1. Write your code so it can run in serial: perfect that first.
2. Deal with communication, synchronization and deadlock on *smaller* number of MPI processes/threads.
3. Only then try full size.

Parallel debugging demands specialized tools: ddt

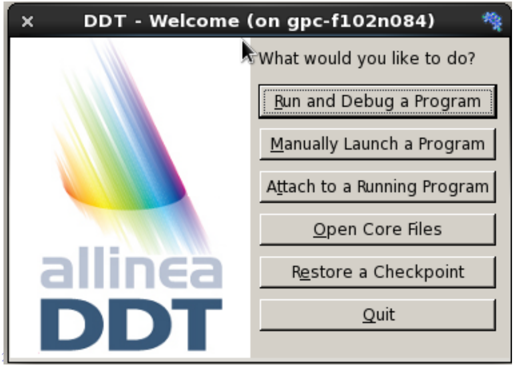
DDT

DDT

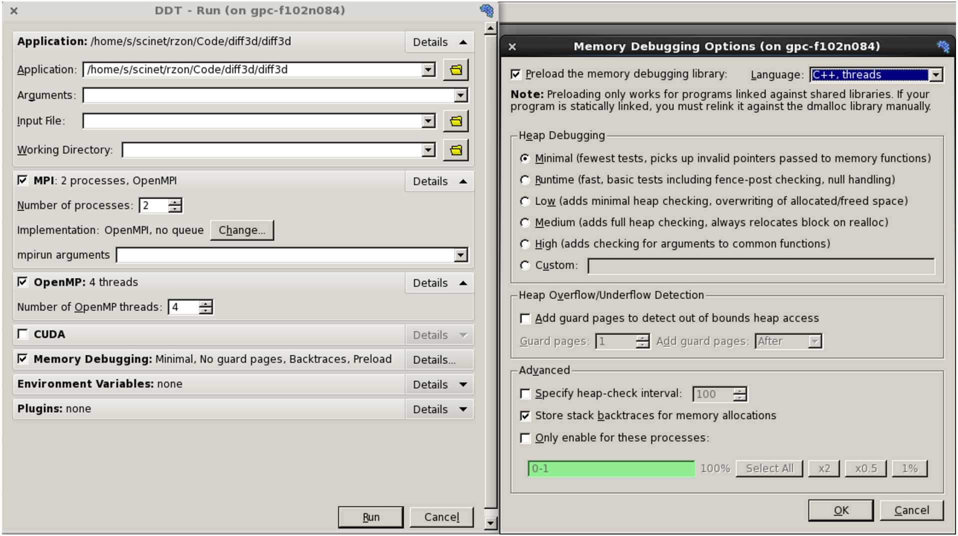
- "Distributed Debugging Tool"
- Powerful GUI-based commercial debugger by *Allinea*.
- Supports C, C++ and Fortran
- Supports MPI, OpenMP, threads, CUDA and more
- Available on all SciNet clusters (GPC, TCS, ARC, P7)
- Available on SHARCNET's kraken, requin, orca and monk.

Launching ddt

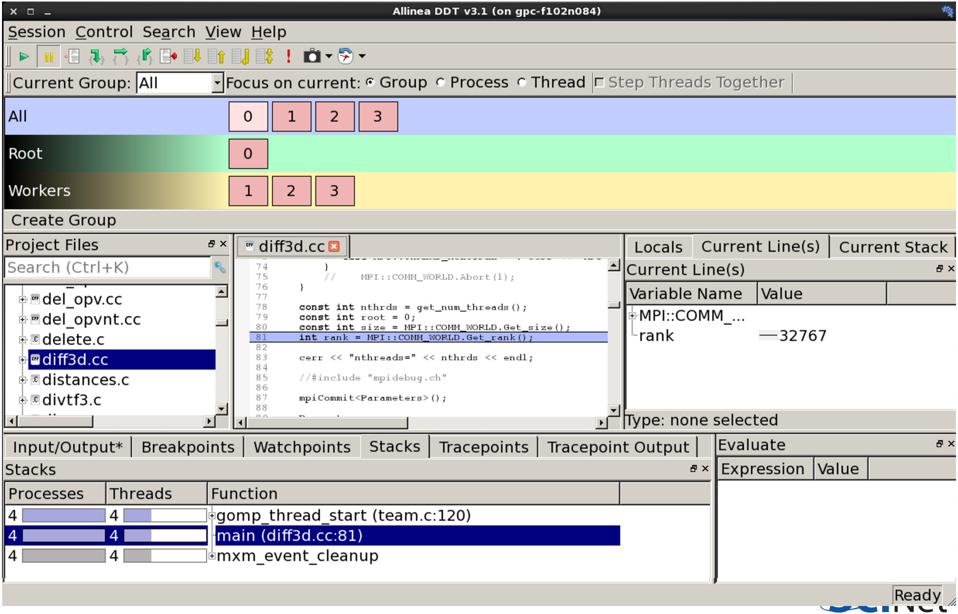
- Load your compiler and MPI modules.
- Load the ddt module: **$ module load ddt**
- Start ddt with one of these:
  $ ddt
  $ ddt <executable compiled with -g flag>
  $ ddt <executable compiled with -g flag> <arguments>
- First time: create config file: OpenMPI (skip other steps)
- Then gui for setting up debug session.



Run and Debug a Program (session setup)

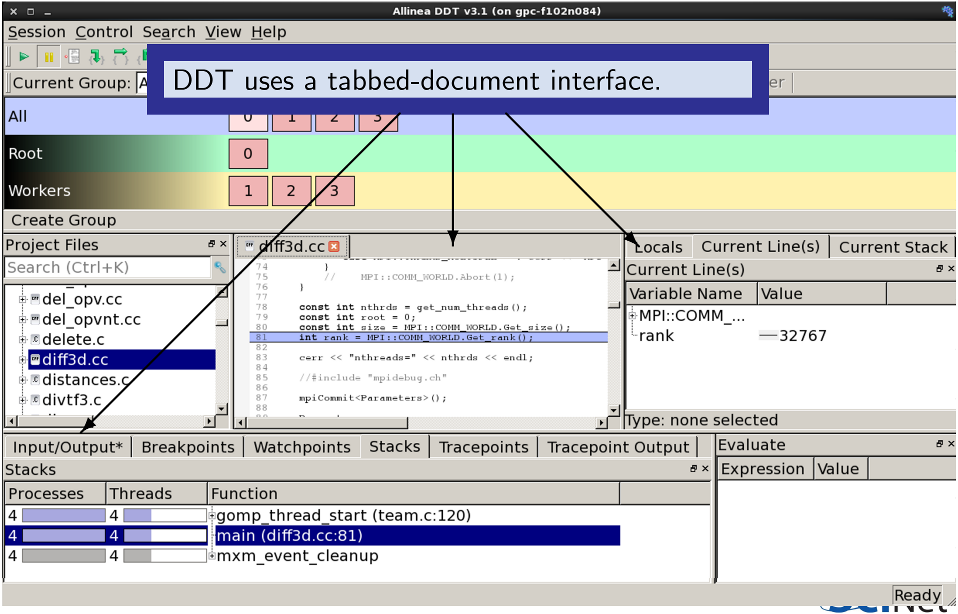


User interface (1)

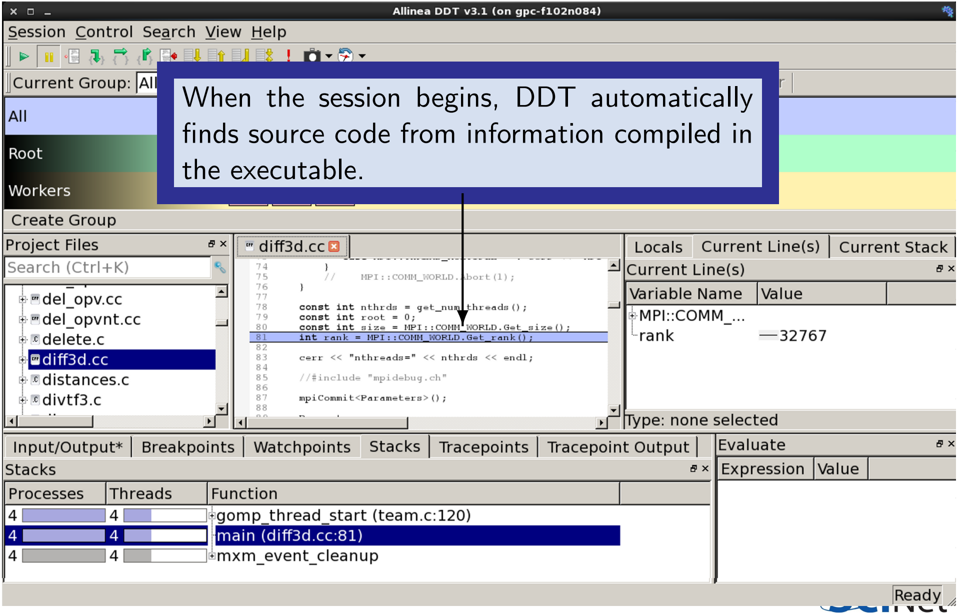


User interface (2)

DDT uses a tabbed-document interface.



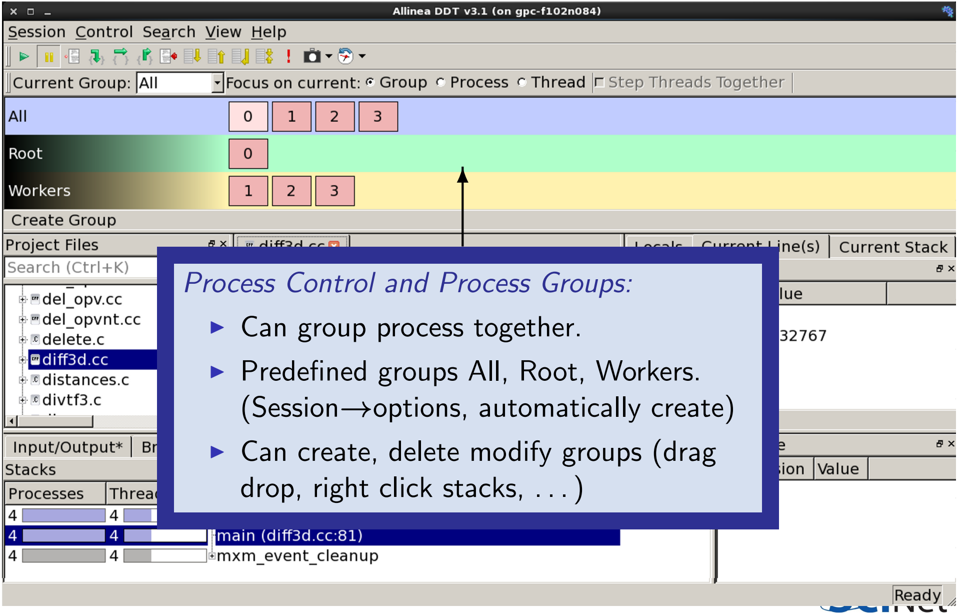
User interface (3)



When the session begins, DDT automatically finds source code from information compiled in the executable.

compute • calcul CANADA

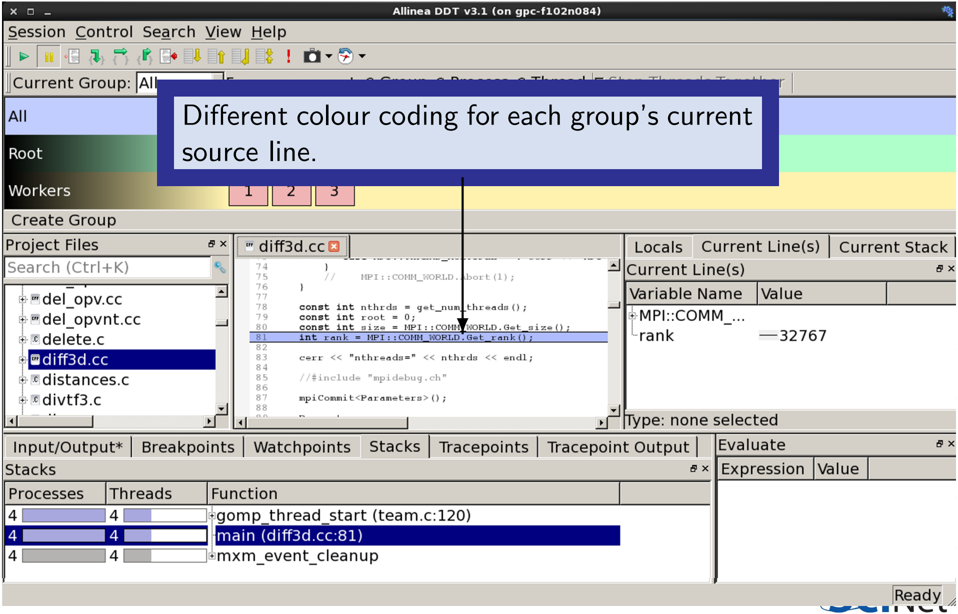
User interface (4)



Process Control and Process Groups:

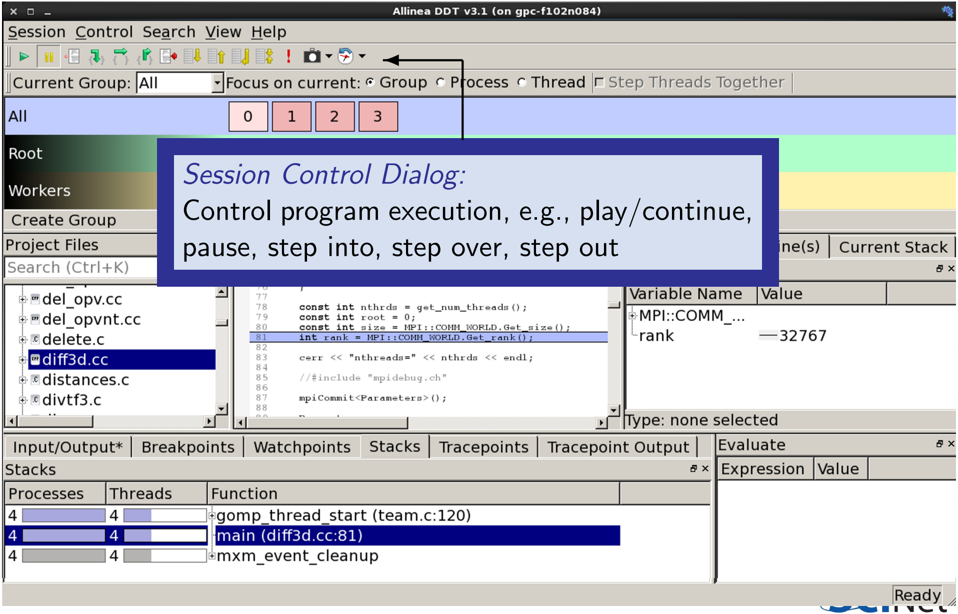
- Can group process together.
- Predefined groups All, Root, Workers. (Session→options, automatically create)
- Can create, delete modify groups (drag drop, right click stacks, . . .)

User interface (5)



Different colour coding for each group's current source line.

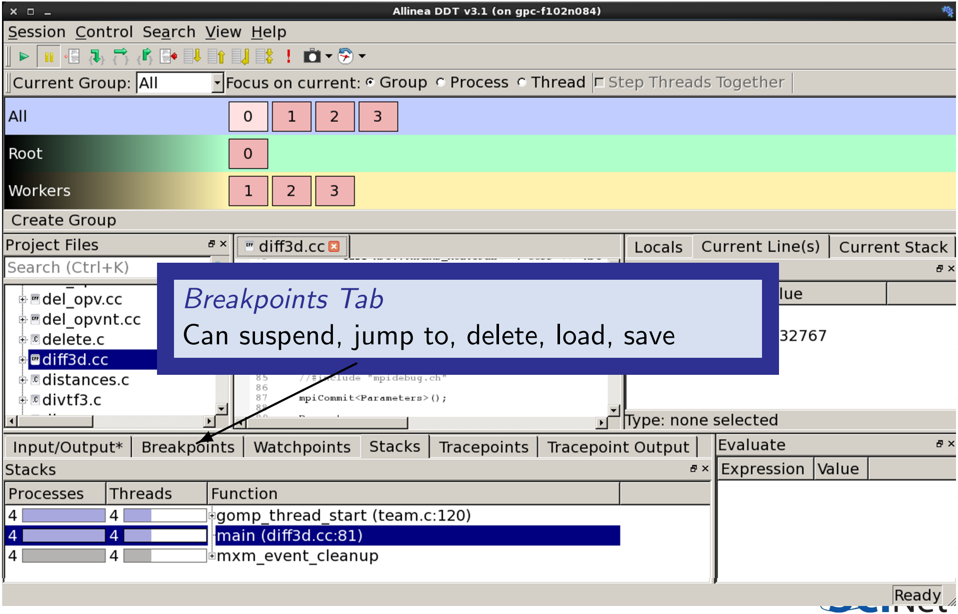
User interface (6)



Session Control Dialog:

Control program execution, e.g., play/continue, pause, step into, step over, step out

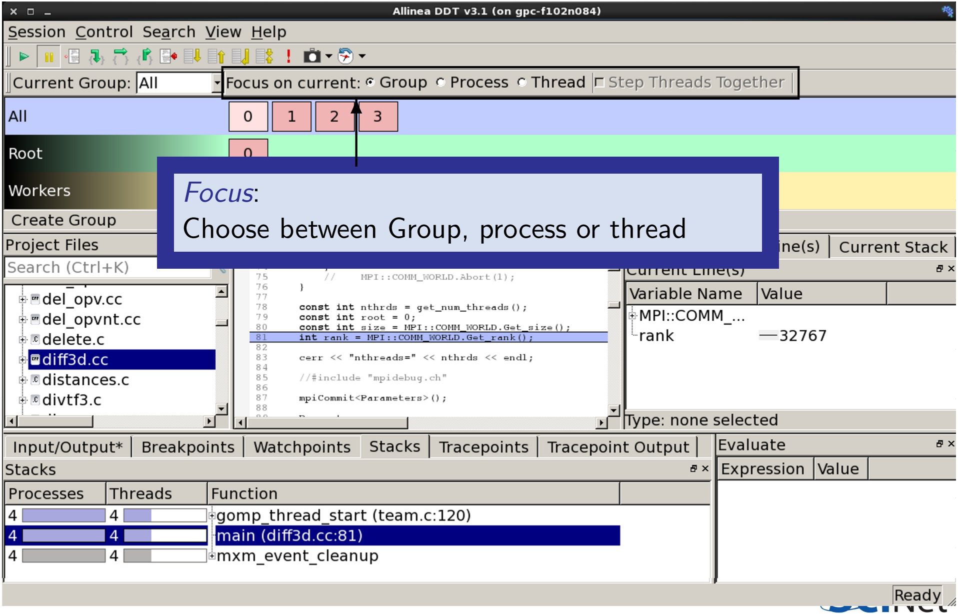
User interface (7)



Breakpoints Tab

Can suspend, jump to, delete, load, save

User interface (8)



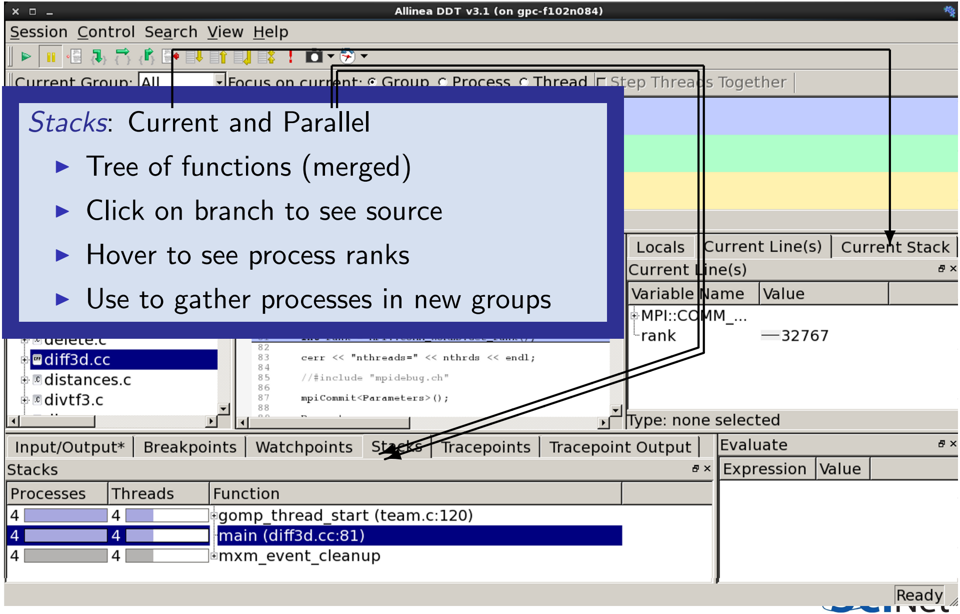
Focus:

Choose between Group, process or thread

User interface (9)

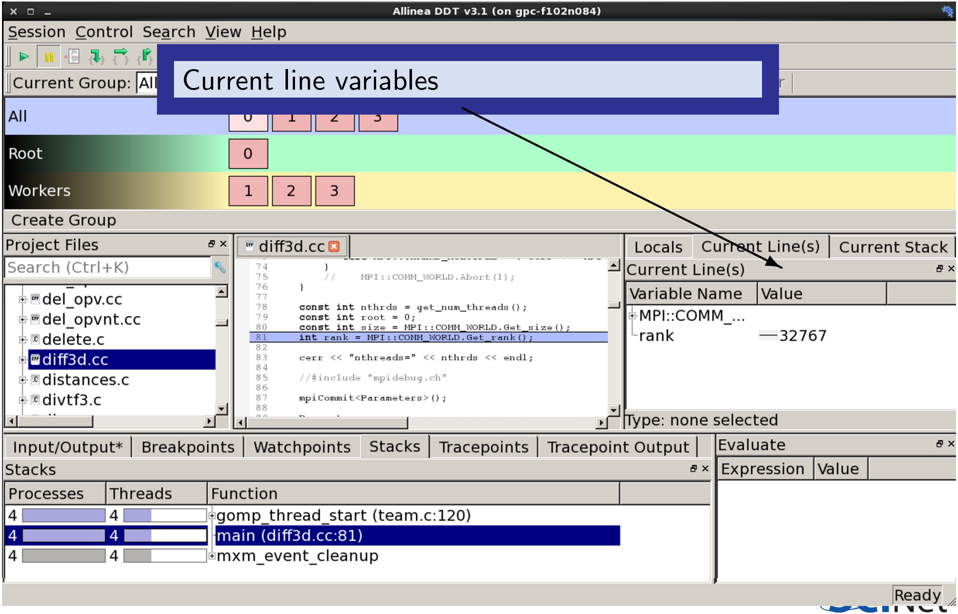
Stacks: Current and Parallel

- Tree of functions (merged)
- Click on branch to see source
- Hover to see process ranks
- Use to gather processes in new groups

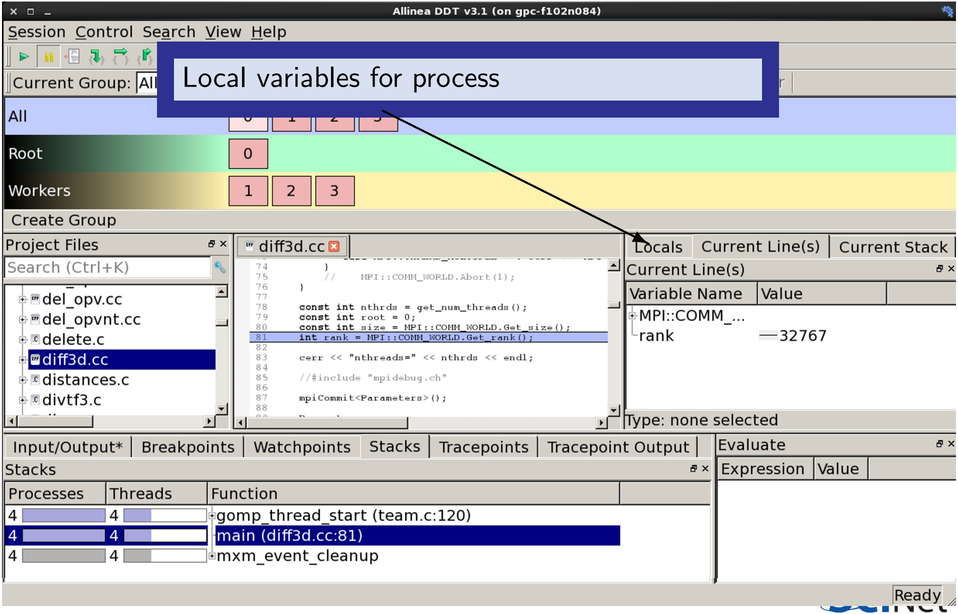


compute • calcul CANADA

User interface (10)

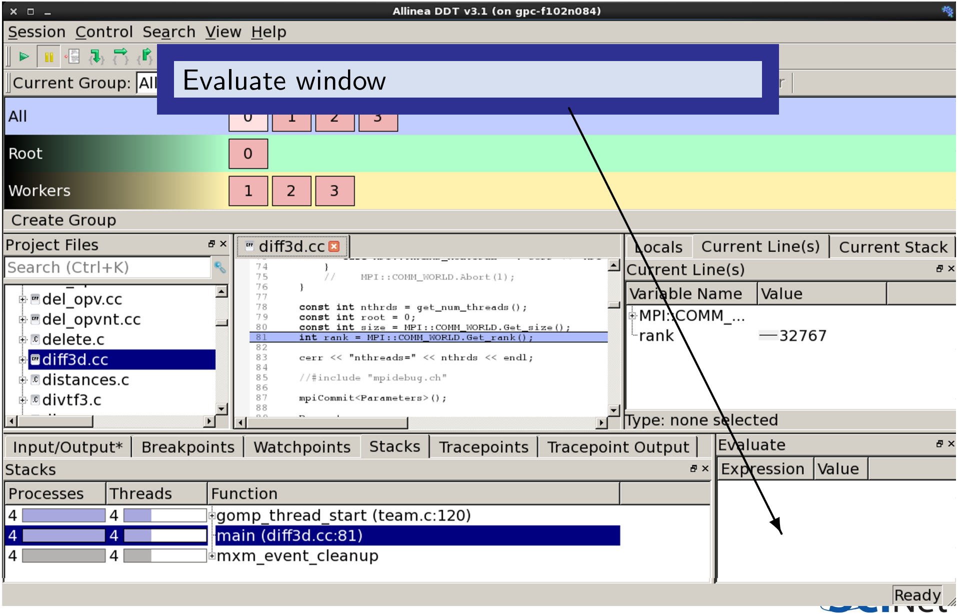


User interface (11)



compute • calcul CANADA

User interface (12)



Demonstration DDT

```
$ cd $SCRATCH/ss2013/HPC106_debug/code
$ source setup
$ cd ex2
$ make
$ ddt ex2
```

Other features of DDT (1)

- Some of the user-modified parameters and windows are saved by right-clicking and selecting a save option in the corresponding window (Groups; Evaluations)
- DDT can load and save sessions.
- *Find* and *Find in Files* in the Search menu.
- *Goto line* in Search menu (or Ctrl-G)
- Synchronize processes in group: Right-click, “Run to here”.
- View multiple source codes simultaneously: Right-click, “Split”
- Right-click power!

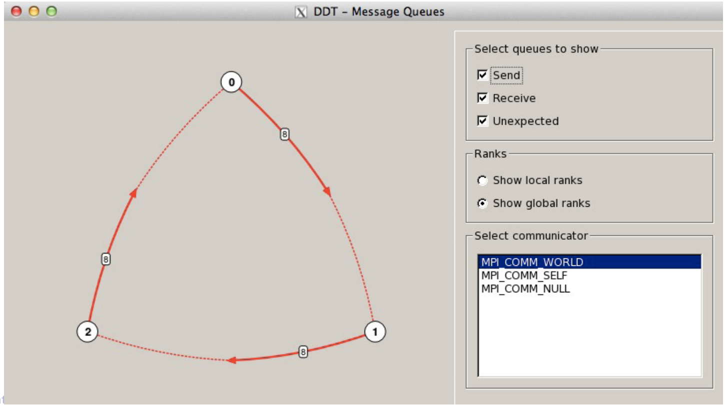
Other features of DDT (2)

- Signal handling: SEGV, FPE, PIPE,ILL
- Support for Fortran modules
- Change data values in evaluate window
- Examine pointers (vector, reference, dereference)
- Multi-dimensional arrays
- Viewer

Other features of DDT (3)

Message Queue

- View → show message queue
- produces both a graphical view and table for active communications
- Helps to find e.g. deadlocks



Demonstration DDT

```
$ cd $SCRATCH/ss2013/HPC106_debug/code
$ source setup
$ cd ex3
$ make
$ ddt ex3
```

Other features of DDT (4)

Memory debugging

- Select “memory debug” in Run window
- Stops on error (before crash or corruption)
- Check pointer (right click in evaluate)
- View, overall memory stats

Demonstration DDT

```
$ cd $SCRATCH/ss2013/HPC106_debug/code
$ source setup
$ cd ex4
$ make
$ ddt ex4
```

Useful references

- G Wilson
  Software Carpentry software-carpentry.org/3_0/debugging.html
- N Matloff and PJ Salzman
  The Art of Debugging with GDB, DDD and Eclipse
- *GDB:* sources.redhat.com/gdb
- *DDT:* www.allinea.com/products/ddt-support
- *SciNet Wiki:* wiki.scinethpc.ca: Tutorials & Manuals