Debugging with GDB and DDT

Ramses van Zon
SciNet HPC Consortium
University of Toronto

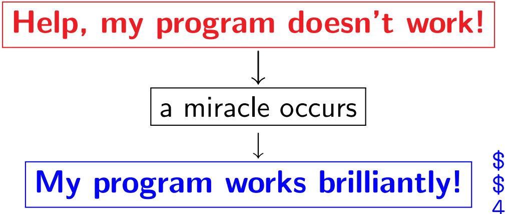
June 28, 2012

Outline

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

Debugging basics

Debugging basics



Help, my program doesn't work!

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

a miracle occurs

My program works brilliantly!

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

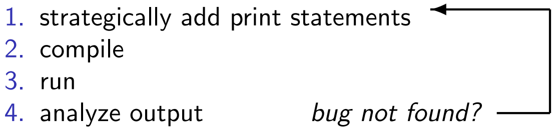
Ways to debug

- Command-line based, symbolic debuggers
  - GNU debugger: *gdb*
  - Intel debugger command-line: *idbc*
- Symbolic debuggers with Graphical User Interface
  - GNU data display debugger: *ddd*
  - Intel debugger: *idb*
  - IDEs: Eclipse, NetBeans (neither on SciNet), *emacs/gdb*
  - Allinea DDT: *ddt*
  - Rogue Wave TotalView (not available at SciNet)

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?

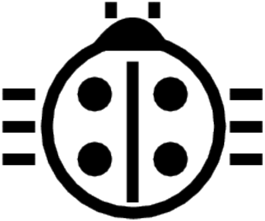
- Removing the extra code after the bug is fixed
- Repeat for each bug

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):
  - Some graphical debuggers slow (connection)
  - Command-line based debuggers fast (esp. gdb).
  - Ddt: gui-based, with graphics light enough to work remotely.
- Graphical and text-based debuggers use the same concepts.

Symbolic debuggers

Preparing the executable

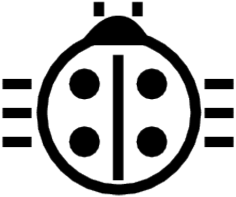
- Required: compile with -g.
- Optional: switch off optimization -O0
- Same for gcc, g++, gfortran, icc, ifort, xlf, mpif90, mpicc, . . .
- For nvcc (i.e. cuda), also add -G

Command-line based symbolic debuggers

- gdb ← **Focus on this one**
- idbc ← **Has gdb mode**

```
$ module load intel
$ icc -g -O0 example.c -o example
$ module load gdb
$ gdb example
...
(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.

GDB command summary

help	h	print description of command
run	r	run from the start (+args)
backtrace/where	ba	function call stack
break	b	set breakpoint
delete	d	delete breakpoint
continue	c	continue
step	s	step into function
next	n	continue until next line
print	p	print variable
quit	q	quit
finish	fin	continue until function end
set variable	set var	change variable
down	do	go to called function
tbreak	tb	set temporary breakpoint
until	unt	continue until line/function
up	up	go to caller
watch	wa	stop if variable changes
watch	wa	stop if variable changes
quit	q	quit gdb

GDB basic building blocks

GDB building blocks #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.

GDB building blocks #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 blocks #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 blocks #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 blocks #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

Graphical symbolic debuggers

Graphical symbolic debuggers

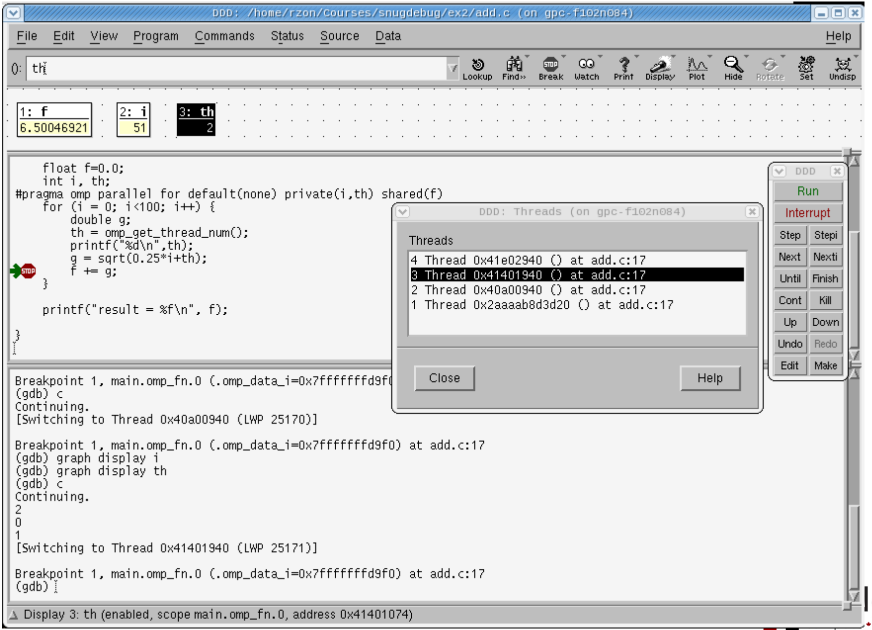
Features

- Nice, more intuitive graphical user interface
- Front to command-line based tools: Same concepts
- Need graphics support (`qsub -X -I ...`)

Available on SciNet

- ddd
  ```
  $ module load gcc ddd
  $ ddd <executable compiled with -g flag>
  ```
- idb
  ```
  $ module load intel java
  ```
  Java slow remotely
  ```
  $ idb <executable compiled with -g flag>
  ```
- ddt
  ```
  $ module load ddt
  (more later)
  ```

Graphical symbolic debuggers - ddd



```
1: f
6.50046921
```

```
2: i
51
```

```
3: th
2
```

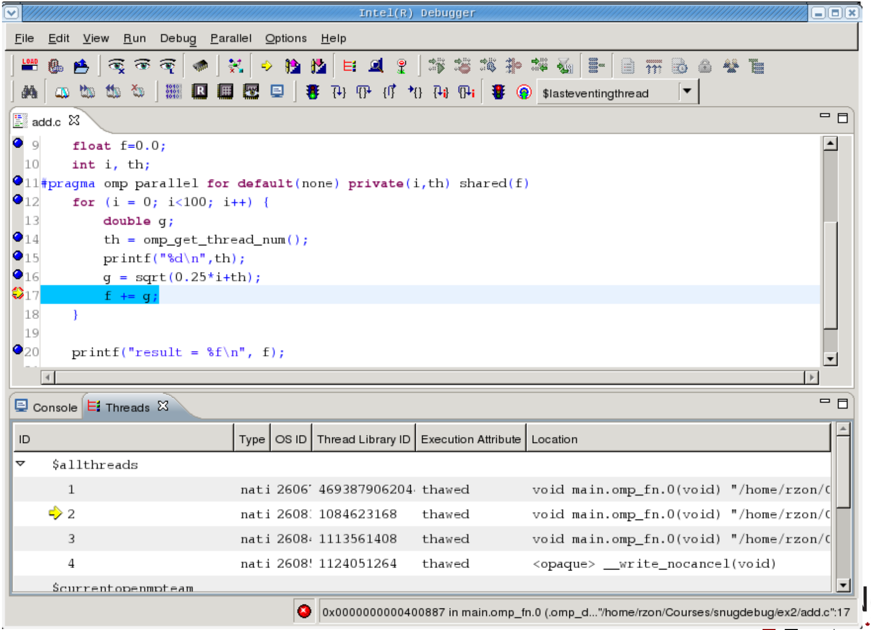
```
    float f=0.0;
    int i, th;
#pragma omp parallel for default(none) private(i,th) shared(f)
    for (i = 0; i<100; i++) {
        double g;
        th = omp_get_thread_num();
        printf("%d\n",th);
        g = sqrt(0.25*i+th);
        f += g;
    }

    printf("result = %f\n", f);

}
```

```
4 Thread 0x41e02940 () at add.c:17
3 Thread 0x41401940 () at add.c:17
2 Thread 0x40a00940 () at add.c:17
1 Thread 0x2aaaab8d3d20 () at add.c:17
```

```
Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0
(gdb) c
Continuing.
[Switching to Thread 0x40a00940 (LWP 25170)]

Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0) at add.c:17
(gdb) graph display i
(gdb) graph display th
(gdb) c
Continuing.
2
0
1
[Switching to Thread 0x41401940 (LWP 25171)]

Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffffd9f0) at add.c:17
(gdb)
```

```
Display 3: th (enabled, scope main.omp_fn.0, address 0x41401074)
```

Graphical symbolic debuggers - idb

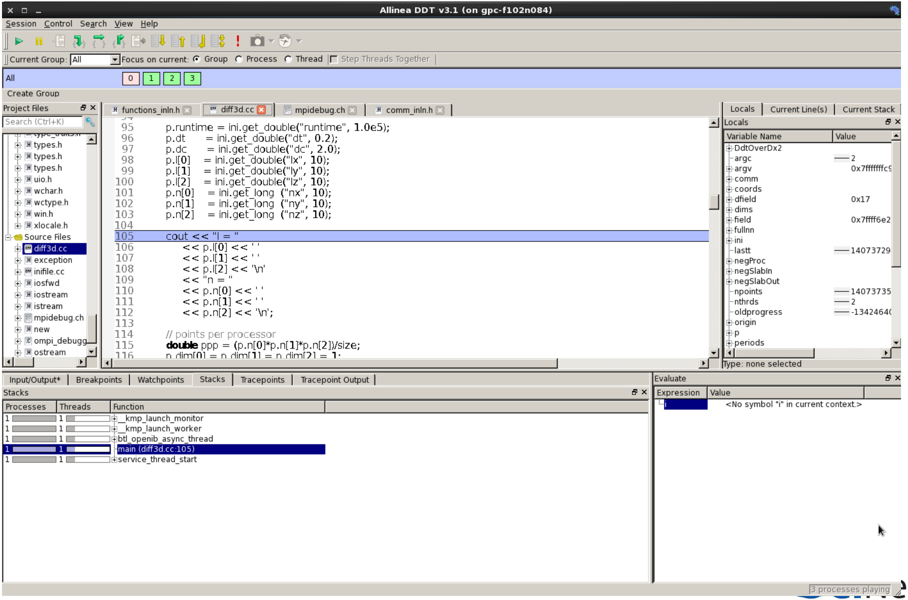


```
    float f=0.0;
    int i, th;
#pragma omp parallel for default(none) private(i,th) shared(f)
    for (i = 0; i<100; i++) {
        double g;
        th = omp_get_thread_num();
        printf("%d\n",th);
        g = sqrt(0.25*i+th);
        f += g;
    }

    printf("result = %f\n", f);
```

ID	Type	OS ID	Thread Library ID	Execution Attribute	Location
$allthreads					
1	nati	2606'	469387906204	thawed	void main.omp_fn.0(void) "/home/rzon/(
2	nati	2608:	1084623168	thawed	void main.omp_fn.0(void) "/home/rzon/(
3	nati	2608	1113561408	thawed	void main.omp_fn.0(void) "/home/rzon/(
4	nati	2608!	1124051264	thawed	<opaque> __write_nocancel(void)
$currentopenmpteam					

0x0000000000400887 in main.omp_fn.0 (.omp_d..."/home/rzon/Courses/snugdebug/ex2/add.c":17

Graphical symbolic debuggers - ddt



```
    p.runtime = ini.get_double("runtime", 1.0e5);
    p.dt      = ini.get_double("dt", 0.2);
    p.dc      = ini.get_double("dc", 2.0);
    p.l[0]    = ini.get_double("lx", 10);
    p.l[1]    = ini.get_double("ly", 10);
    p.l[2]    = ini.get_double("lz", 10);
    p.n[0]    = ini.get_long  ("nx", 10);
    p.n[1]    = ini.get_long  ("ny", 10);
    p.n[2]    = ini.get_long  ("nz", 10);

    cout << "l = "
         << p.l[0] << ' '
         << p.l[1] << ' '
         << p.l[2] << '\n'
         << "n = "
         << p.n[0] << ' '
         << p.n[1] << ' '
         << p.n[2] << '\n';

    // points per processor
    double ppp = (p.n[0]*p.n[1]*p.n[2])/size;
    p.dim[0] = p.dim[1] = p.dim[2] = 1;
```

Locals

Variable Name	Value
DdtOverDx2	
argc	2
argv	0x7fffffffc9
comm	
coords	
dfield	0x17
dims	
field	0x7ffff6e2
fullnn	
ini	
lastt	14073729
negProc	
negSlabIn	
negSlabOut	
npoints	14073735
nthrds	2
oldprogress	-1342464(
origin	
p	
periods	

Type: none selected

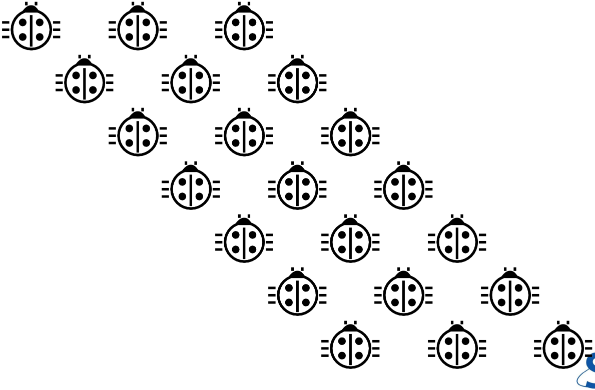
Stacks

Processes	Threads	Function
1	1	__kmp_launch_monitor
1	1	__kmp_launch_worker
1	1	btl_openib_async_thread
1	1	main (diff3d.cc:105)
1	1	service_thread_start

Evaluate

Expression	Value
i	<No symbol "i" in current context.>

Parallel debugging



Parallel debugging

- Challenge: Simultaneous execution
- Shared memory:
  OpenMP (Open Multi-Processing)
  pthreads (POSIX threads)
  - Private/shared variables

    Intel compiler extra flag: `-debug parallel`
  - Race conditions
- Distributed memory:
  MPI (Message Passing Interface)
  - Communication
  - Deadlock
- Hard to solve: some commercial debuggers do a good job.
  But let's see how the command-line ones handle it.

Parallel debugging - 1 Shared memory

Use gdb for

- Track each thread's execution and variables
- OpenMP serialization: `p omp_set_num_threads(1)`
- Step 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 - 2 Distributed memory

Advanced gdb (not recommended!)

- You want #proc terminals with gdb for each process?
- Possible, but brace yourself!
- Small number of procs:
  1. Start terminals: by default X forwarding from compute nodes
  2. Submit your job on scinet
  3. Make sure its runs: checkjob -v
  4. From each terminal, ssh into the appropriate nodes
  5. Do `top` or `ps -C <exe>` to find process id (pid)
  6. Attach debugger with `gdb -pid <pid>`.
  7. This will interrupt the process.

Parallel debugging - 2 Distributed memory

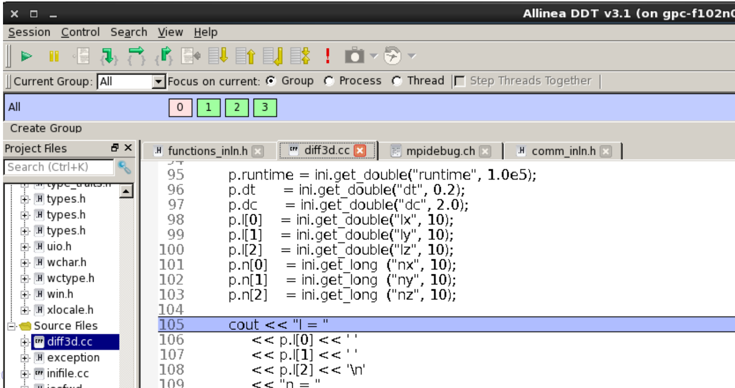
Advanced tricks

Wait, so the program started already?

- Yes, and that's probably not what you want.
- Instead, put infinite loop into your code:
  ```
  int j=1;
  while(j) sleep(5);
  ```
- Once attached, go "up" until at while loop.
- do "set var j=0"
- now you can step, continue, etc.

Now let's take a look at DDT...

DDT



```
        p.runtime = ini.get_double("runtime", 1.0e5);
        p.dt      = ini.get_double("dt", 0.2);
        p.dc      = ini.get_double("dc", 2.0);
        p.l[0]    = ini.get_double("lx", 10);
        p.l[1]    = ini.get_double("ly", 10);
        p.l[2]    = ini.get_double("lz", 10);
        p.n[0]    = ini.get_long  ("nx", 10);
        p.n[1]    = ini.get_long  ("ny", 10);
        p.n[2]    = ini.get_long  ("nz", 10);

        cout << "l = "
            << p.l[0] << ' '
            << p.l[1] << ' '
            << p.l[2] << '\n'
            << "n = "
```

DDT

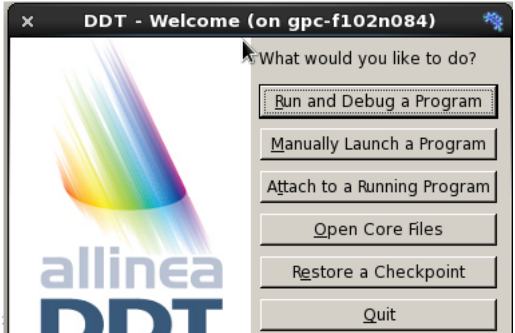
- “Distributed Debugging Tool”
- Powerful GUI-based commercial debugger by *Allinea*.
- Supports C, C++ and Fortran
- Supports MPI, OpenMP, threads, CUDA and more
- Available on all SciNet clusters (GPC, TCS, ARC, P7)
- Available on SHARCNET’s kraken, requin, orca and monk.

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>
```

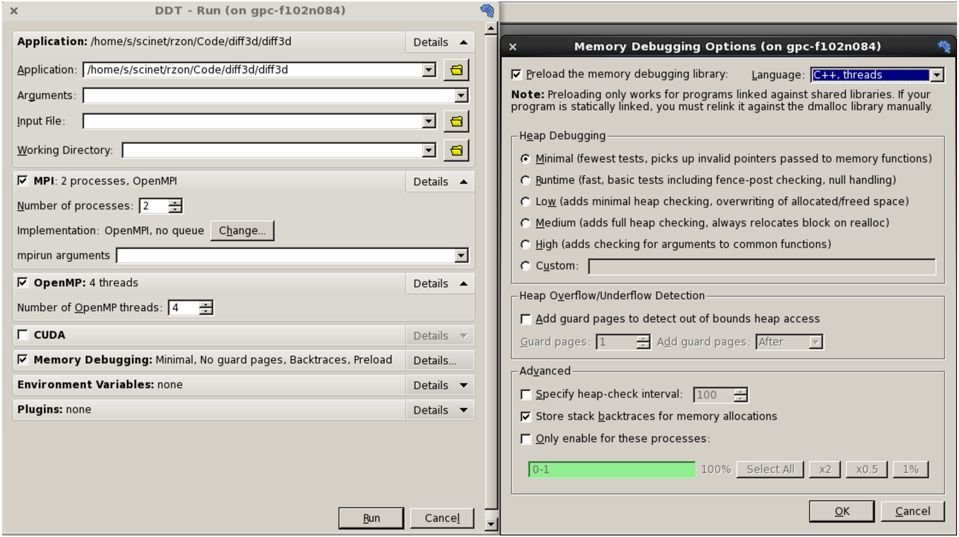
- First time: create config file: OpenMPI (skip other steps)
- Then gui for setting up debug session.



DDT - Welcome (on gpc-f102n084)

What would you like to do?

Run and Debug a Program (session setup)



DDT - Run (on gpc-f102n084)

Application: /home/s/scinet/rzon/Code/diff3d/diff3d

Application: /home/s/scinet/rzon/Code/diff3d/diff3d

Arguments:

Input File:

Working Directory:

☑ **MPI**: 2 processes, OpenMPI

Number of processes: 2

Implementation: OpenMPI, no queue

mpirun arguments

☑ **OpenMP:** 4 threads

Number of OpenMP threads: 4

☐ **CUDA**

☑ **Memory Debugging:** Minimal, No guard pages, Backtraces, Preload

Environment Variables: none

Plugins: none

Memory Debugging Options (on gpc-f102n084)

☑ Preload the memory debugging library: Language: C++, 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

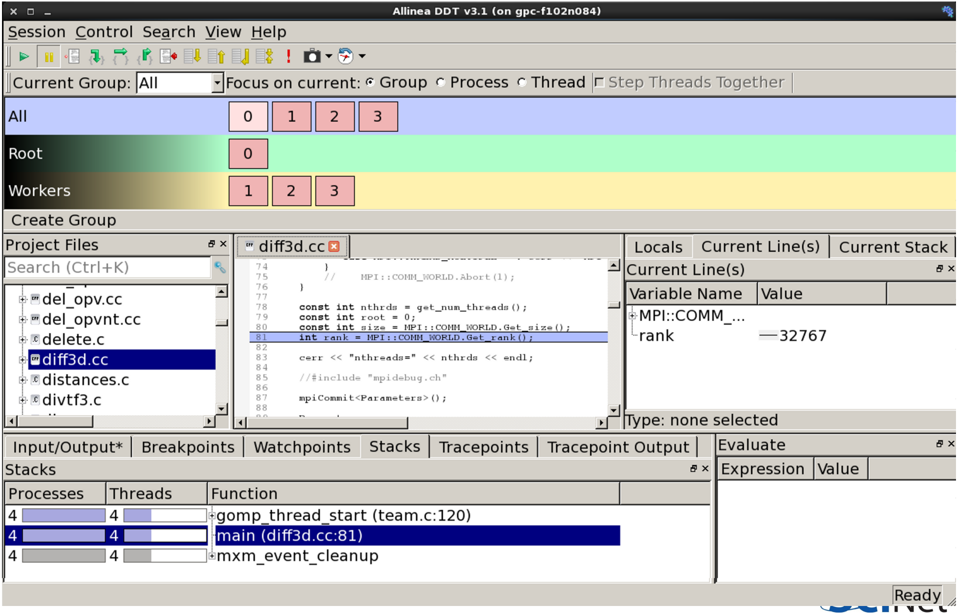
☐ Specify heap-check interval: 100

☑ Store stack backtraces for memory allocations

☐ Only enable for these processes:

0-1 100%

User interface (1)



```
        }
        //     MPI::COMM_WORLD.Abort(1);
    }

    const int nthrds = get_num_threads();
    const int root = 0;
    const int size = MPI::COMM_WORLD.Get_size();
    int rank = MPI::COMM_WORLD.Get_rank();

    cerr << "nthreads=" << nthrds << endl;

    //#include "mpidebug.ch"

    mpiCommit<Parameters>();
```

Current Line(s)

Variable Name	Value
MPI::COMM_...	
rank	32767

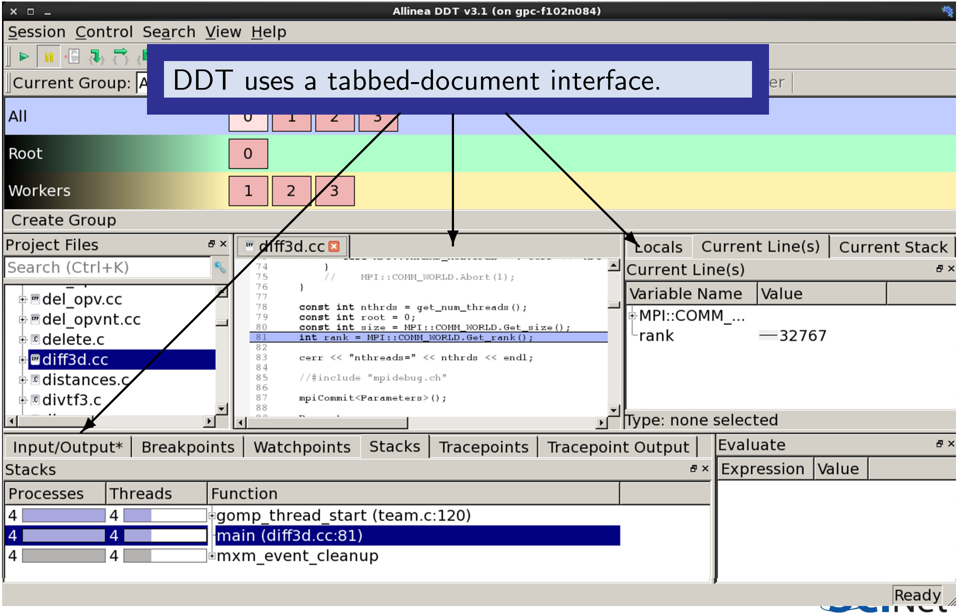
Type: none selected

Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

Expression	Value

User interface (2)



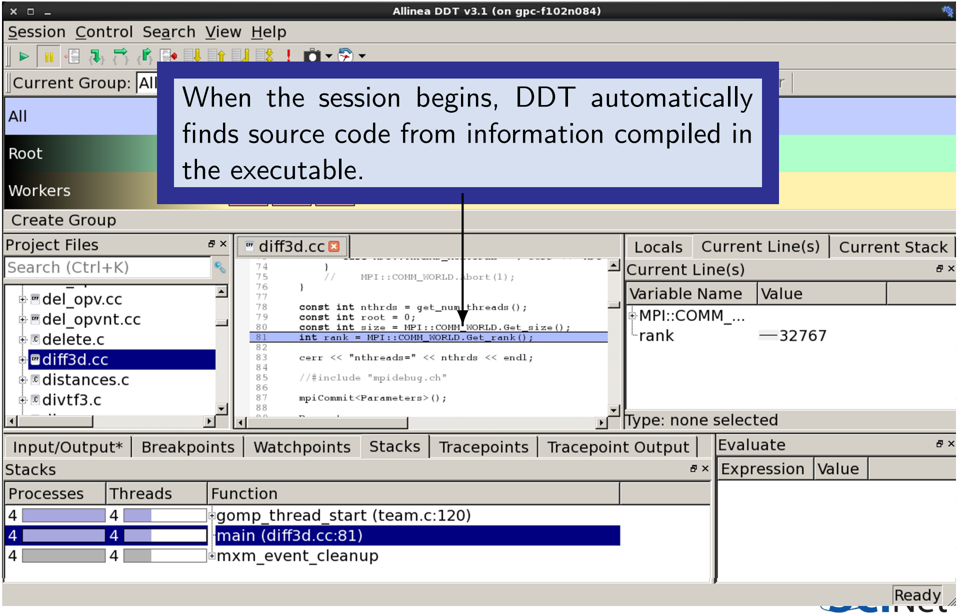
DDT uses a tabbed-document interface.

```
        }
        //     MPI::COMM_WORLD.Abort(1);
    }

    const int nthrds = get_num_threads();
    const int root = 0;
    const int size = MPI::COMM_WORLD.Get_size();
    int rank = MPI::COMM_WORLD.Get_rank();

    cerr << "nthreads=" << nthrds << endl;

    //#include "mpidebug.ch"

    mpiCommit<Parameters>();
```

Variable Name	Value
MPI::COMM_...	
rank	32767

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

User interface (3)



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

```
            }
            //      MPI::COMM_WORLD.Abort(1);
        }

        const int nthrds = get_num_threads();
        const int root = 0;
        const int size = MPI::COMM_WORLD.Get_size();
        int rank = MPI::COMM_WORLD.Get_rank();

        cerr << "nthreads=" << nthrds << endl;

        //#include "mpidebug.ch"

        mpiCommit<Parameters>();
```

Current Line(s)

Variable Name	Value
MPI::COMM_...	
rank	32767

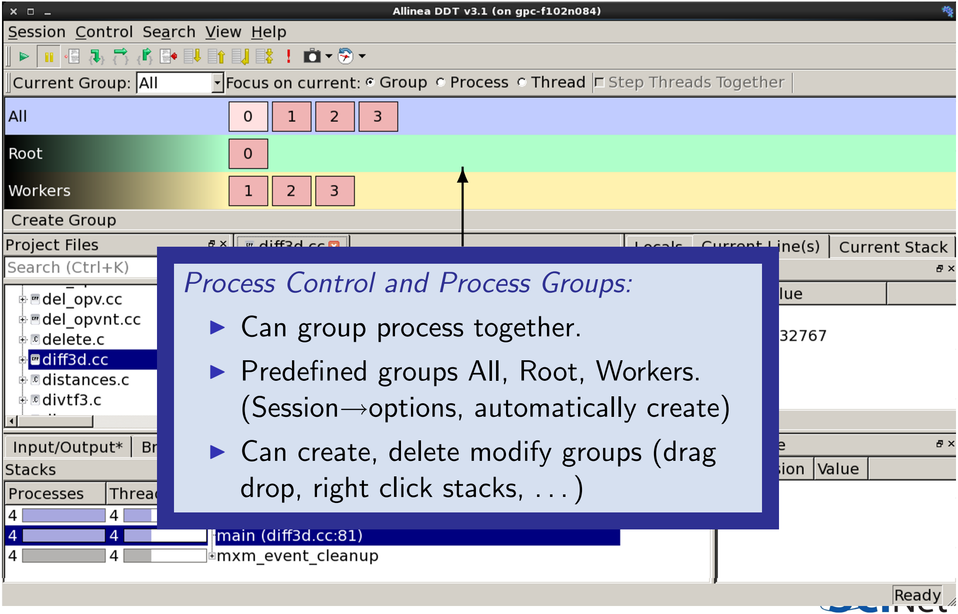
Type: none selected

Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

Expression	Value

User interface (4)



Current Group: All

Focus on current: ● Group ○ Process ○ Thread □ Step Threads Together

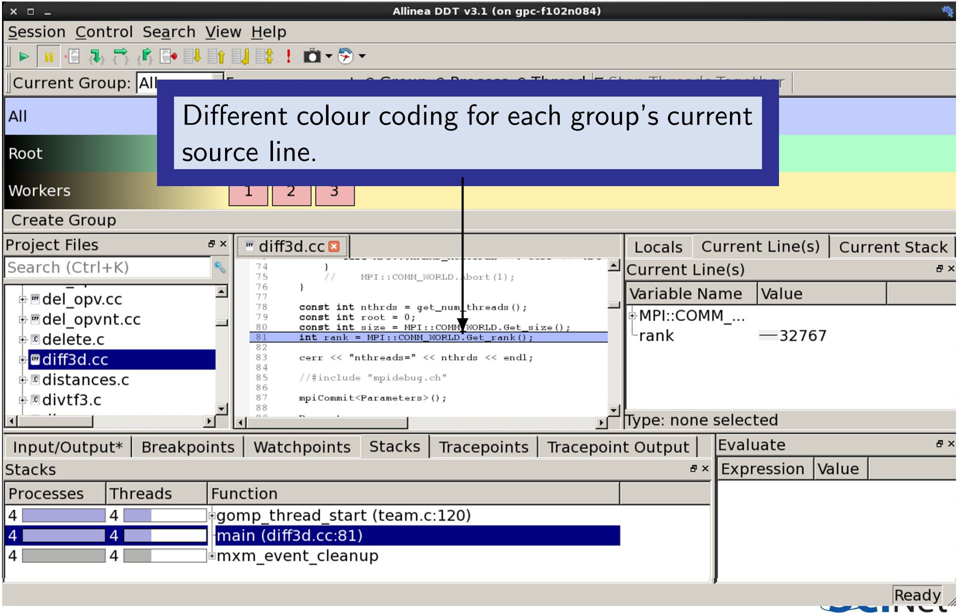
Processes	Threads	
4	4	
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

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.

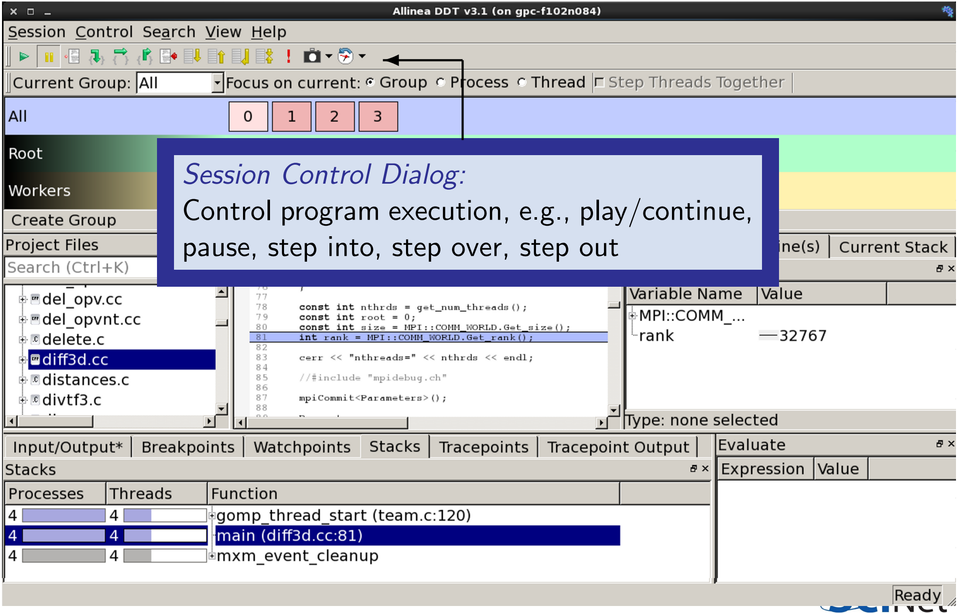


```
        }
        //     MPI::COMM_WORLD.Abort(1);
    }

    const int nthrds = get_num_threads();
    const int root = 0;
    const int size = MPI::COMM_WORLD.Get_size();
    int rank = MPI::COMM_WORLD.Get_rank();

    cerr << "nthreads=" << nthrds << endl;

    //#include "mpidebug.ch"

    mpiCommit<Parameters>();
```

Variable Name	Value
MPI::COMM_...	
rank	32767

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

User interface (6)



Session Control Dialog:

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

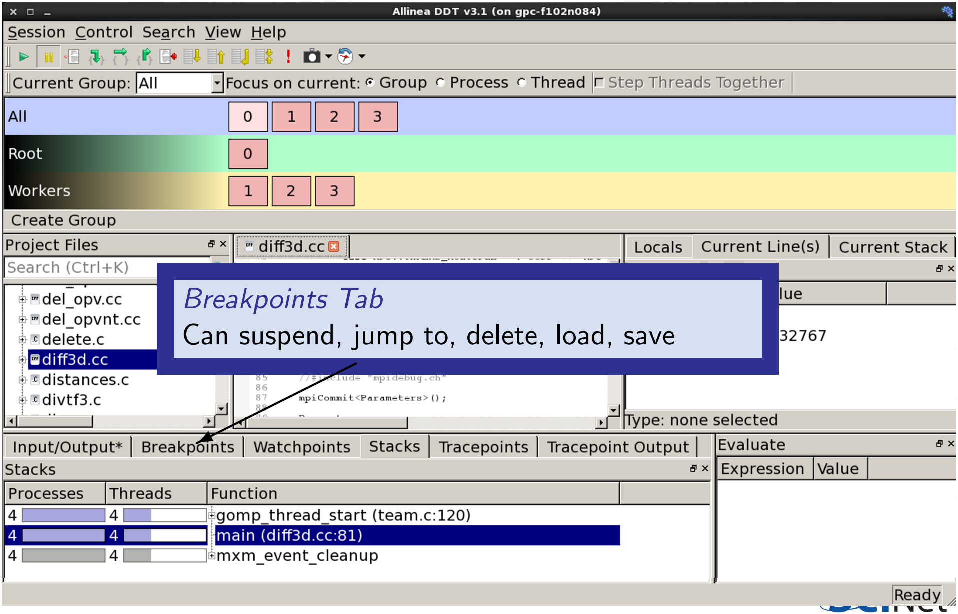
```
      const int nthrds = get_num_threads();
      const int root = 0;
      const int size = MPI::COMM_WORLD.Get_size();
      int rank = MPI::COMM_WORLD.Get_rank();

      cerr << "nthreads=" << nthrds << endl;

      //#include "mpidebug.ch"

      mpiCommit<Parameters>();
```

Variable Name	Value
MPI::COMM_...	
rank	32767

Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

User interface (7)



Current Group: All Focus on current: ● Group ○ Process ○ Thread □ Step Threads Together

All	0 1 2 3
Root	0
Workers	1 2 3

```
85      //#include "mpidebug.ch"
86
87      mpiCommit<Parameters>();
```

Type: none selected

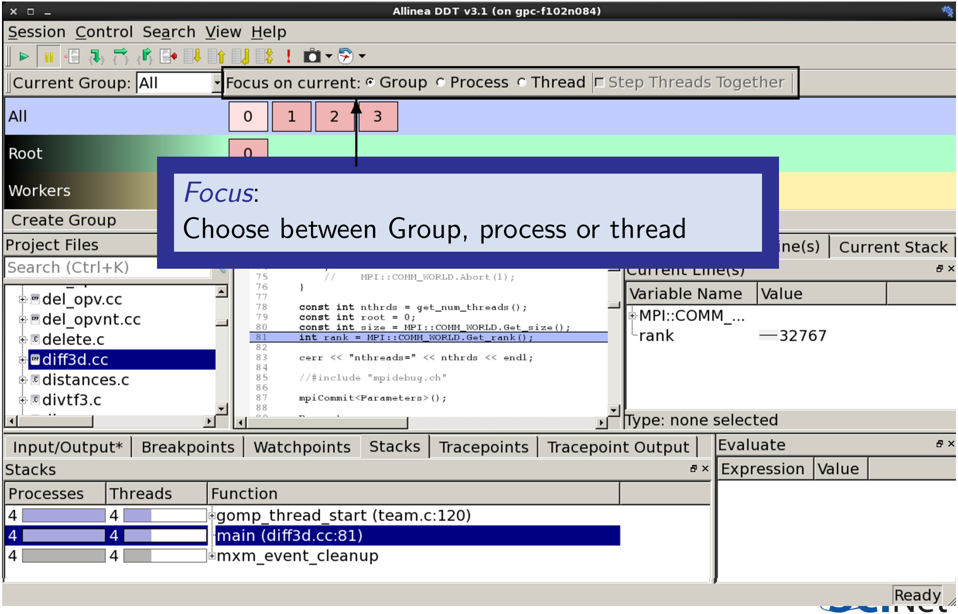
Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

Breakpoints Tab

Can suspend, jump to, delete, load, save

User interface (8)



```
        //      MPI::COMM_WORLD.Abort(1);
    }

    const int nthrds = get_num_threads();
    const int root = 0;
    const int size = MPI::COMM_WORLD.Get_size();
    int rank = MPI::COMM_WORLD.Get_rank();

    cerr << "nthreads=" << nthrds << endl;

    //#include "mpidebug.ch"

    mpiCommit<Parameters>();
```

Variable Name	Value
MPI::COMM_...	
rank	32767

Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

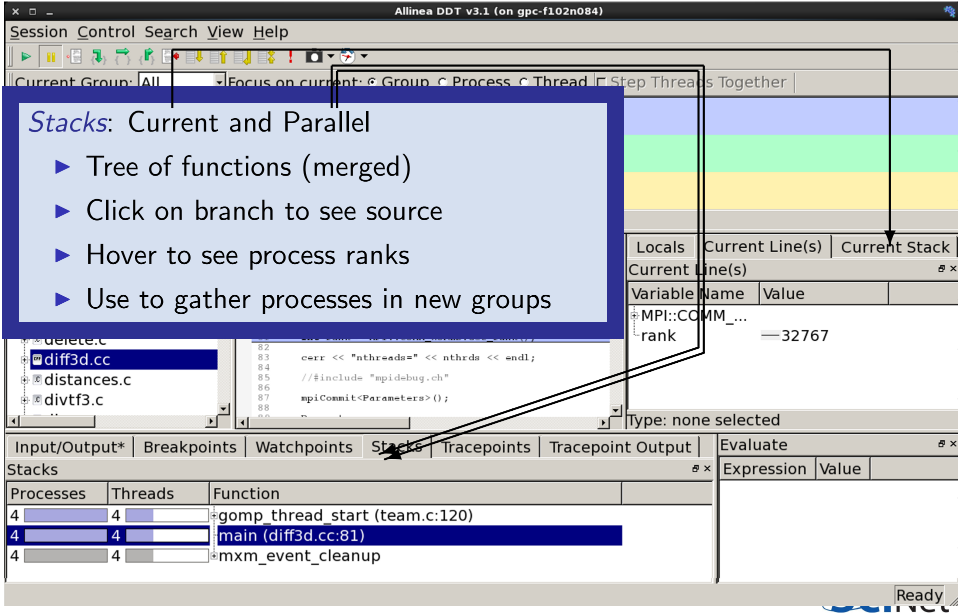
Focus:

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



```
    cerr << "nthreads=" << nthrds << endl;

    //#include "mpidebug.ch"

    mpiCommit<Parameters>();
```

Current Line(s)

Variable Name	Value
MPI::COMM_...	
rank	32767

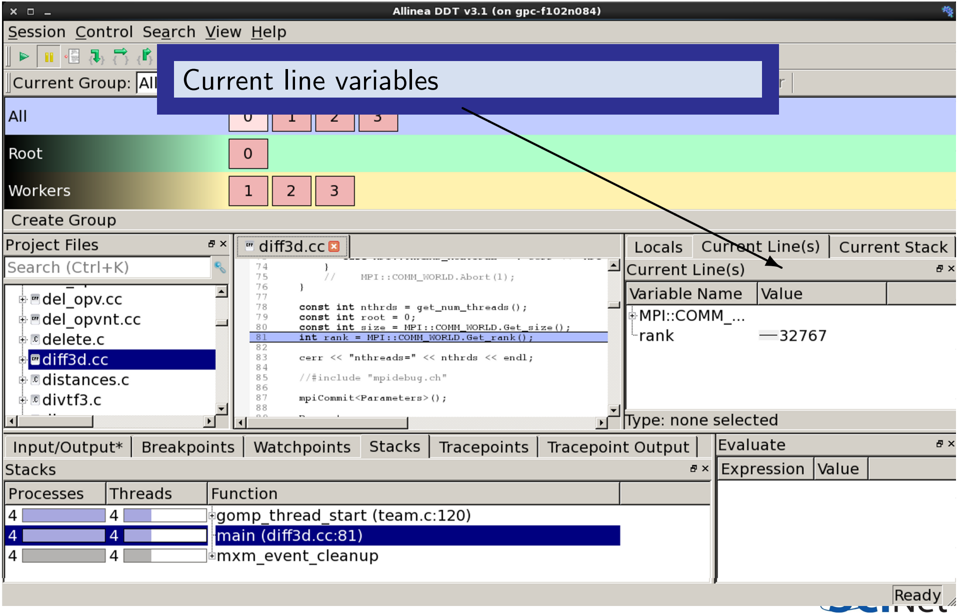
Type: none selected

Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

Expression	Value

User interface (10)



Current line variables

```
        }
        //     MPI::COMM_WORLD.Abort(1);
    }

    const int nthrds = get_num_threads();
    const int root = 0;
    const int size = MPI::COMM_WORLD.Get_size();
    int rank = MPI::COMM_WORLD.Get_rank();

    cerr << "nthreads=" << nthrds << endl;

    //#include "mpidebug.ch"

    mpiCommit<Parameters>();
```

Current Line(s)

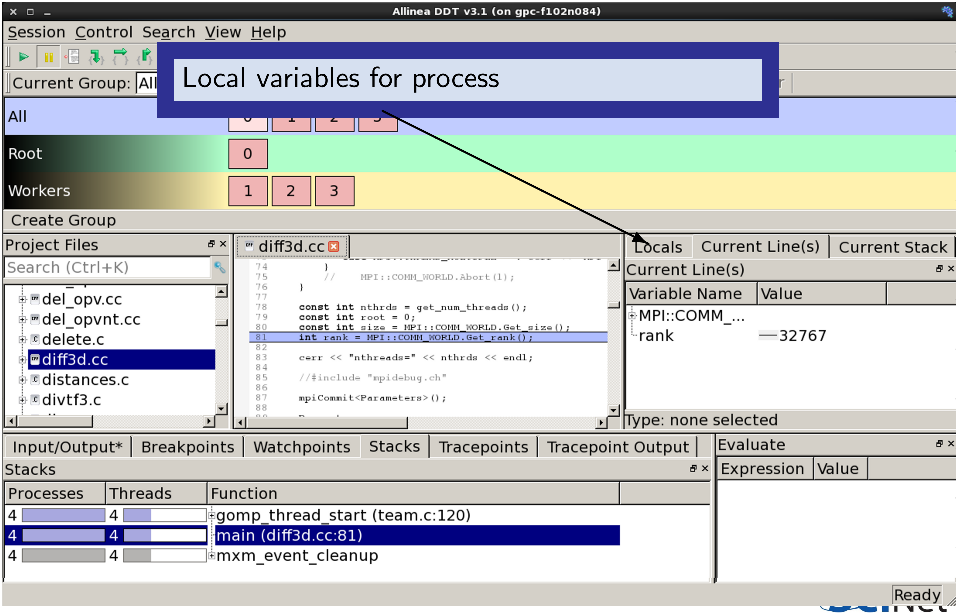
Variable Name	Value
MPI::COMM_...	
rank	32767

Type: none selected

Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

User interface (11)



Local variables for process

```
        }
        //     MPI::COMM_WORLD.Abort(1);
    }

    const int nthrds = get_num_threads();
    const int root = 0;
    const int size = MPI::COMM_WORLD.Get_size();
    int rank = MPI::COMM_WORLD.Get_rank();

    cerr << "nthreads=" << nthrds << endl;

    //#include "mpidebug.ch"

    mpiCommit<Parameters>();
```

Current Line(s)

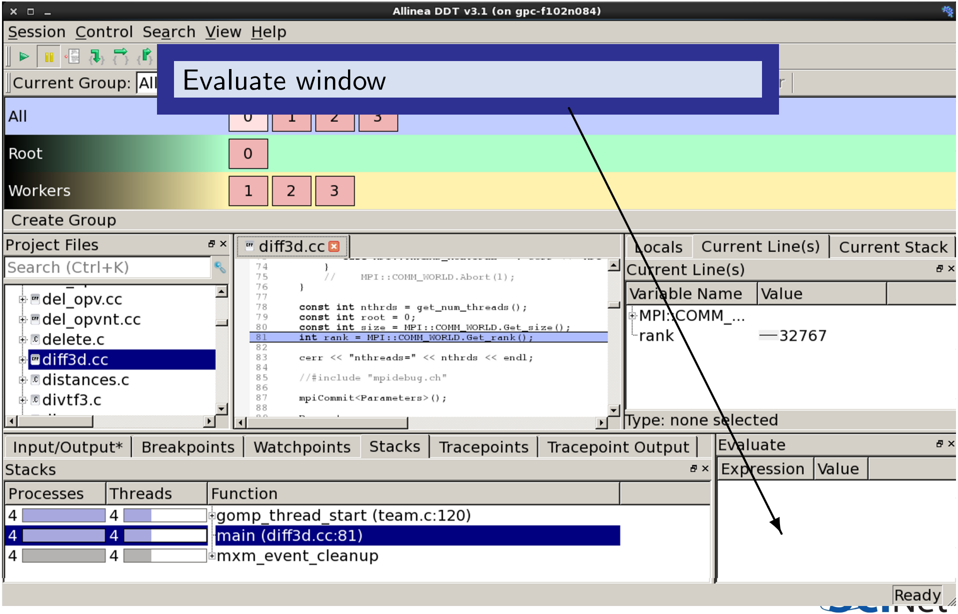
Variable Name	Value
MPI::COMM_...	
rank	32767

Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

Expression	Value

User interface (12)



Evaluate window

All	0	1	2	3
Root	0			
Workers	1	2	3	

```
        }
    //      MPI::COMM_WORLD.Abort(1);
    }

    const int nthrds = get_num_threads();
    const int root = 0;
    const int size = MPI::COMM_WORLD.Get_size();
    int rank = MPI::COMM_WORLD.Get_rank();

    cerr << "nthreads=" << nthrds << endl;

    //#include "mpidebug.ch"

    mpiCommit<Parameters>();
```

Current Line(s)

Variable Name	Value
MPI::COMM_...	
rank	32767

Stacks

Processes	Threads	Function
4	4	gomp_thread_start (team.c:120)
4	4	main (diff3d.cc:81)
4	4	mxm_event_cleanup

Evaluate

Expression	Value

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!

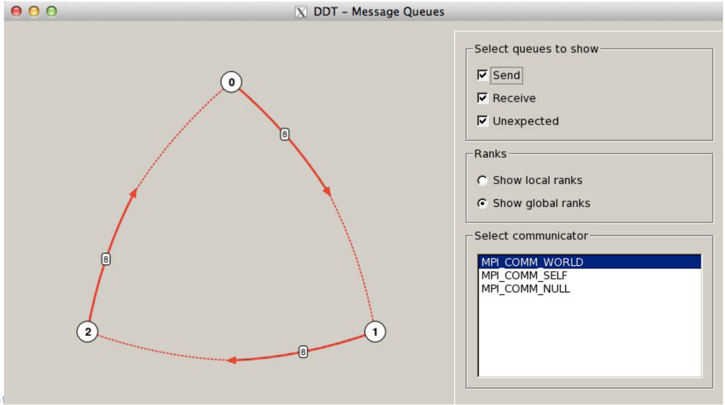
Other features of DDT (2)

- 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



DDT – Message Queues

Select queues to show

☑ Send

☑ Receive

☑ Unexpected

Ranks

○ Show local ranks

● Show global ranks

Select communicator

MPI_COMM_WORLD
MPI_COMM_SELF
MPI_COMM_NULL

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

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