

# Parallel Debugging with DDT

Ramses van Zon, Jonathan Dursi  
SciNet HPC Consortium  
University of Toronto

November 28, 2012



# Outline

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

# Debugging basics



# Debugging basics

# Debugging basics

**Help, my program doesn't work!**

# Debugging basics

**Help, my program doesn't work!**

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

# Debugging basics

**Help, my program doesn't work!**

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

a miracle occurs

# Debugging basics

**Help, my program doesn't work!**



a miracle occurs



**My program works brilliantly!**

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



# 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
```

# 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 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

# Common symptoms

## Errors at compile time

- ▶ Syntax errors: easy to fix
- ▶ Library issues
- ▶ Cross-compiling
- ▶ Compiler warnings

# 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!**

# 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!

# 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



# 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 ( $\text{sqrt}(-0.0)$ ), 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

# Ways to debug

- ▶ Preemptive:
  - ▶ Turn on compiler warnings: fix or understand them!
  - ▶ Check your assumptions (e.g. use assert).

# 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!

# 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

# 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

# Ways to debug

- ▶ Command-line based, symbolic debuggers
  - ▶ GNU debugger: *[gdb](#)*
  - ▶ Intel debugger command-line: *[idbc](#)*

# 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

# What's wrong with using print statements?

## Strategy

- ▶ Constant cycle:

# What's wrong with using print statements?

## Strategy

- ▶ Constant cycle:
  1. strategically add print statements

# What's wrong with using print statements?

## Strategy

- ▶ Constant cycle:
  1. strategically add print statements
  2. compile

# What's wrong with using print statements?

## Strategy

- ▶ Constant cycle:
  1. strategically add print statements
  2. compile
  3. run



# What's wrong with using print statements?

## Strategy

- ▶ Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output

# 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?*

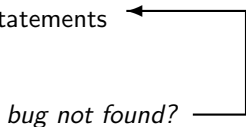
# What's wrong with using print statements?

## Strategy

► Constant cycle:

1. strategically add print statements
2. compile
3. run
4. analyze output

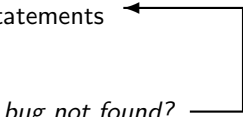
*bug not found?*



# What's wrong with using print statements?

## Strategy

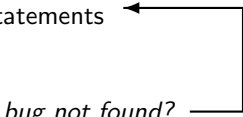
- ▶ Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output

*bug not found?* 
- ▶ Removing the extra code after the bug is fixed

# What's wrong with using print statements?

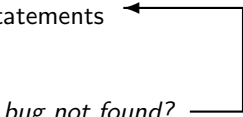
## Strategy

- ▶ 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

# What's wrong with using print statements?

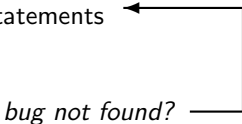
## Strategy

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

## Problems with this approach

# What's wrong with using print statements?

## Strategy

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

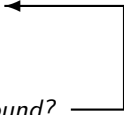
## Problems with this approach

- ▶ Time consuming
- ▶ Error prone
- ▶ Changes memory, timing...

# What's wrong with using print statements?

## Strategy

- ▶ Constant cycle:
  1. strategically add print statements
  2. compile
  3. run
  4. analyze output

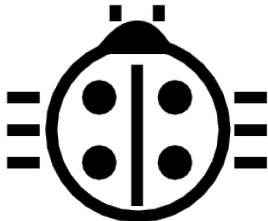
*bug not found?* 
- ▶ Removing the extra code after the bug is fixed
- ▶ Repeat for each bug

## Problems with this approach

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



## Symbolic debuggers



# Symbolic debuggers

## Features

# Symbolic debuggers

## Features

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

# 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?

# 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`

# 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`

# 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`



# 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**

# Symbolic debuggers

## Preparing the executable

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

## Command-line based symbolic debuggers

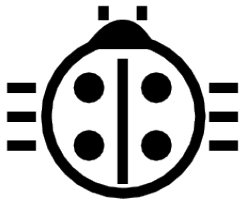
- ▶ `gdb` ← **Focus on this one**
- ▶ `idbc` ← **Has gdb mode**

```
$ 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

# GDB building blocks #1: Inspect crashes

## Inspecting core files

**Core** = file containing state of program after a crash



# 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

# 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?

# 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 Ctrl-C while program is running in gdb
- ▶ gdb will show you where the program was.

# GDB building blocks #2: Function call stack

## Interrupting program

- ▶ Press Ctrl-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 building blocks #2: Function call stack

## Interrupting program

- ▶ Press Ctrl-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

<code>backtrace</code>	function call stack
<code>continue</code>	continue
<code>down</code>	go to called function
<code>up</code>	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 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

<code>list</code>	list part of code
<code>next</code>	continue until next line
<code>step</code>	step into function
<code>finish</code>	continue until function end
<code>until</code>	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.

# 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

<code>delete</code>	<code>unset breakpoint</code>
<code>condition</code>	<code>break if condition met</code>
<code>disable</code>	<code>disable breakpoint</code>
<code>enable</code>	<code>enable breakpoint</code>
<code>info breakpoints</code>	<code>list breakpoints</code>
<code>tbreak</code>	<code>temporary breakpoint</code>

# 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 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

<code>print</code>	<code>print variable</code>
<code>display</code>	<code>print at every prompt</code>
<code>set variable</code>	<code>change variable</code>
<code>watch</code>	<code>stop if variable changes</code>

# Demonstration GDB



## 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 ...`)

# 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

The screenshot displays the DDD (Data Display Debugger) graphical user interface. The main window shows a C program with OpenMP parallelism. The code is as follows:

```
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);
```

The interface includes a menu bar (File, Edit, View, Program, Commands, Status, Source, Data, Help) and a toolbar with icons for various debugging actions. A status bar at the bottom indicates the current display: "Display 3: th (enabled, scope main.omp\_fn.0, address 0x41401074)".

A "Threads" window is open, showing the following threads:

Thread ID	Address
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

A control panel on the right side of the window contains the following buttons: Run, Interrupt, Step (Stepi), Next (Nexti), Until (Finish), Cont (Kill), Up (Down), Undo (Redo), Edit (Make), and Close (Help).

The main window also shows a GDB console with the following output:

```
Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffff9f0)
(gdb) c
Continuing.
[Switching to Thread 0x40a00940 (LWP 25170)]

Breakpoint 1, main.omp_fn.0 (.omp_data_i=0x7fffffff9f0) 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=0x7fffffff9f0) at add.c:17
(gdb) |
```

# Graphical symbolic debuggers - idb

The screenshot shows the Intel(R) Debugger interface. The main window displays the source code of a C program named `add.c`. The code is as follows:

```
9 float f=0.0;
10 int i, th;
11 #pragma omp parallel for default(none) private(i,th) shared(f)
12 for (i = 0; i<100; i++) {
13     double g;
14     th = omp_get_thread_num();
15     printf("%d\n",th);
16     g = sqrt(0.25*i+th);
17     f += g;
18 }
19
20 printf("result = %f\n", f);
```

The line `f += g;` is highlighted in blue. Below the source code, the `Threads` window is open, showing a table of threads:

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

The status bar at the bottom indicates the current thread is `0x0000000000400887` in `main.omp_fn.0` at `"/home/rzon/Courses/snugdebug/ex2/add.c":17`.

# Graphical symbolic debuggers - ddt

The screenshot displays the Alinea DDT v3.1 interface. The main window shows a C++ source file named `diff3d.cc` with the following code:

```
95 p.runtime = ini.get_double("runtime", 1.0e5);
96 p.dt = ini.get_double("dt", 0.2);
97 p.dc = ini.get_double("dc", 2.0);
98 p.l[0] = ini.get_double("lx", 10);
99 p.l[1] = ini.get_double("ly", 10);
100 p.l[2] = ini.get_double("lz", 10);
101 p.n[0] = ini.get_long("nx", 10);
102 p.n[1] = ini.get_long("ny", 10);
103 p.n[2] = ini.get_long("nz", 10);
104
105 cout << "l = "
106 << p.l[0] << " "
107 << p.l[1] << " "
108 << p.l[2] << "\n"
109 << "n = "
110 << p.n[0] << " "
111 << p.n[1] << " "
112 << p.n[2] << "\n";
113
114 // points per processor
115 double ppp = (p.n[0]*p.n[1]*p.n[2])/size;
116 n.dim[0] = n.dim[1] = n.dim[2] = 1;
```

The interface includes a "Stacks" panel at the bottom left showing the current execution stack:

Processes	Threads	Function
1	1	+ kmp_launch_monitor
1	1	+ kmp_launch_worker
1	1	+bb_openib_async_thread
1	1	main (diff3d.cc:105)
1	1	+service_thread_start

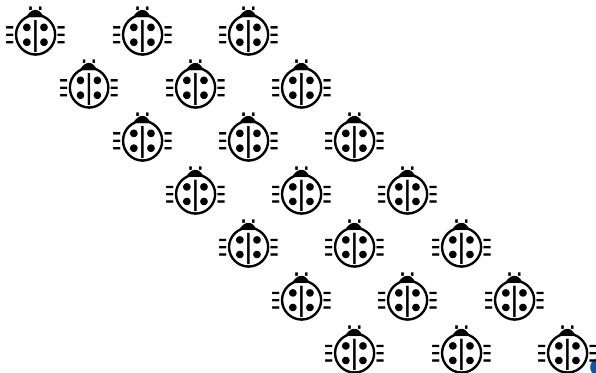
The "Locals" panel on the right shows the current line's local variables:

Variable Name	Value
DdtOverDx2	
-argc	2
-argv	0x7fffff6c
-comm	
-coords	
-dfield	0x17
-dims	
-field	0x7ffff6e2
-fullnn	
-ini	
-lastt	14073729
-negProc	
-negSlabin	
-negSlabOut	
-npoints	14073735
-nthrds	2
-oldprogress	-1342464
-origin	
-p	
-periods	

The "Evaluate" panel at the bottom right shows the current expression and its value:

Expression	Value
<code>*</code>	<No symbol '*' in current context.>

## Parallel debugging



# Parallel debugging

- ▶ Challenge: Simultaneous execution
- ▶ Shared memory:
  - OpenMP* (Open Multi-Processing)
  - threads* (POSIX threads)
    - ▶ Private/shared variables
      - Intel compiler extra flag: `-debug parallel`
      - Later GNU compilers: `-gstabs`
    - ▶ 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>`

# 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.



# 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

The screenshot shows the Allinea DDT v3.1 debugger interface. The title bar reads "Allinea DDT v3.1 (on gpc-f102n)". The menu bar includes "Session", "Control", "Search", "View", and "Help". Below the menu is a toolbar with various debugging icons. The status bar shows "Current Group: All" and "Focus on current: Group Process Thread Step Threads Together".

The "Create Group" section shows a group named "All" with four sub-groups labeled 0, 1, 2, and 3.

The "Project Files" pane on the left shows a tree view of source files, including "types.h", "uio.h", "wchar.h", "wctype.h", "win.h", "xlocale.h", "Source Files", "diff3d.cc", "exception", "infile.cc", and "isofed".

The main editor window displays the source code for "diff3d.cc". The code is as follows:

```
95     p.runtime = ini.get_double("runtime", 1.0e5);
96     p.dt      = ini.get_double("dt", 0.2);
97     p.dc      = ini.get_double("dc", 2.0);
98     p.l[0]    = ini.get_double("lx", 10);
99     p.l[1]    = ini.get_double("ly", 10);
100    p.l[2]    = ini.get_double("lz", 10);
101    p.n[0]    = ini.get_long ("nx", 10);
102    p.n[1]    = ini.get_long ("ny", 10);
103    p.n[2]    = ini.get_long ("nz", 10);
104
105    cout << "l = "
106         << p.l[0] << " "
107         << p.l[1] << " "
108         << p.l[2] << " \n"
109         << "n = "
```



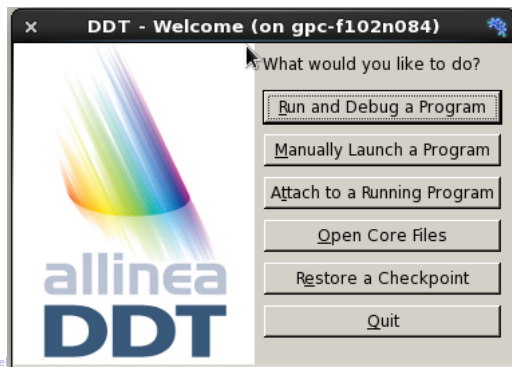
- ▶ “Distributed Debugging Tool”
- ▶ Powerful GUI-based commercial debugger by *Allinea*.
- ▶ Supports C, C++ and Fortran
- ▶ Supports MPI, OpenMP, threads, CUDA and more
- ▶ Available on all SciNet clusters (GPC, TCS, ARC, P7)

## 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.

# 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)

The image shows the DDT - Run dialog box for a program named 'diff3d'. The main configuration panel is on the left, and the 'Memory Debugging Options' panel is open on the right.

**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%



# Run and Debug a Program (session setup)

DDT - Run (on gpc-f102n084)

**Application:** /home/s/scinet/rzon/Code/diff3d/diff3d Details ^

Application:  📁

Arguments:

Input File:  📁

Working Directory:  📁

**MPI:** 2 processes, OpenMPI Details ^

Number of processes:  ⬇️ ⬆️ ⬇️

Implementation: OpenMPI, no queue Change...

mpirun arguments

**OpenMP:** 4 threads Details ^

Number of OpenMP threads:  ⬇️ ⬆️ ⬇️

**CUDA** Details v

**Memory Debugging:** Minimal, No guard pages, Backtraces, Preload Details...

**Environment Variables:** none Details v

**Plugins:** none Details v

### Memory Debugging Options (o

**Preload the memory debugging library:**  **Minimal** (fewest tests, picks up invalid pointers)- Runtime** (fast, basic tests including fence-post)
- Low** (adds minimal heap checking, overwriting)
- Medium** (adds full heap checking, always relo
- High** (adds checking for arguments to commo
- Custom:**

Heap Overflow/Underflow Detection

**Add guard pages to detect out of bounds hea**

Guard pages:  ⬇️ ⬆️ ⬇️ Add guard pages:  **Specify heap-check interval:**  ⬇️ ⬆️ ⬇️

**Store stack backtraces for memory allocations**

**Only enable for these processes:**

100%

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

Details ▲

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

Arguments:

Input File:

Working Directory:

 **MPI:** 2 processes, OpenMPI

Details ▲

Number of processes: 2

Implementation: OpenMPI, no queue

Change...

mpirun arguments

 **OpenMP:** 4 threads

Details ▲

Number of OpenMP threads: 4

 **CUDA**

Details ▼

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

Details...

**Environment Variables:** none

Details ▼

**Plugins:** none

Details ▼

**Memory De** Preload the memory de**Note:** Preloading only works if the program is statically linked

## Heap Debugging

 Minimal (fewest tests, Runtime (fast, basic te Low (adds minimal he Medium (adds full hea High (adds checking f Custom: 

## Heap Overflow/Underflow

 Add guard pages to c

Guard pages: 1

## Advanced

 Specify heap-check in Store stack backtrace Only enable for these

0-1

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

Details ▲

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

Arguments:

Input File:

Working Directory:

 **MPI:** 2 processes, OpenMPI

Details ▲

Number of processes: 2

Implementation: OpenMPI, no queue

Change...

mpirun arguments

 **OpenMP:** 4 threads

Details ▲

Number of OpenMP threads: 4

 **CUDA**

Details ▼

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

Details...

**Environment Variables:** none

Details ▼

**Plugins:** none

Details ▼

 Preload**Note:** Pr  
program

Heap D

 Mini Run Low Med High Cust

Heap O

 Add

Guard p

Advanc

 Spe Sto

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

Details ▲

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

Arguments:

Input File:

Working Directory:

**MPI:** 2 processes, OpenMPI

Details ▲

Number of processes: 2

Implementation: OpenMPI, no queue

Change...

mpirun arguments

**OpenMP:** 4 threads

Details ▲

Number of OpenMP threads: 4

**CUDA**

Details ▼

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

Details...

ff3d Details ▲

f3d ▼ 

▼

▼ 

▼ 

Details ▲

▼

▼

Details ▲

Details ▼

es, Backtraces, Preload Details...

## Memory Debugging Options

Preload the memory debugging library: [ ]

**Note:** Preloading only works for programs link program is statically linked, you must relink it a

### Heap Debugging

Minimal (fewest tests, picks up invalid poin

Runtime (fast, basic tests including fence-p

Low (adds minimal heap checking, overwr

Medium (adds full heap checking, always r

High (adds checking for arguments to com

Custom: [ ]

### Heap Overflow/Underflow Detection

Add guard pages to detect out of bounds

Guard pages: [ 1 ] Add guard pages:

### Advanced



## 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:     Add guard pages: After

### Advanced

# Run and Debug a Program (session setup)

The image shows two overlapping windows from the DDT (Data Display Tool) interface. The background window is titled "DDT - Run (on gpc-f102n084)" and contains configuration options for running a program. The foreground window is titled "Memory Debugging Options (on gpc-f102n084)" and provides settings for memory debugging.

**DDT - Run (on gpc-f102n084)**

- Application: /home/s/scinet/rzon/Code/diff3d/diff3d
- Application: /home/s/scinet/rzon/Code/diff3d/diff3d
- Arguments: [empty]
- Input File: [empty]
- Working Directory: [empty]
- MPI:** 2 processes, OpenMPI
- Number of processes: 2
- Implementation: OpenMPI, no queue
- mpirun arguments: [empty]
- 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: [empty]

**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% Select All x2 x0.5 1%

Buttons: Run, Cancel, OK, Cancel

# User interface (1)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. At the top, the 'Session Control' menu is visible. Below it, the 'Current Group' is set to 'All', and 'Focus on current' is set to 'Group'. The 'Step Threads Together' checkbox is checked. The main area shows a hierarchical view of the process tree with three groups: 'All' (0, 1, 2, 3), 'Root' (0), and 'Workers' (1, 2, 3). The 'Project Files' pane on the left shows a list of files, with 'diff3d.cc' selected. The central editor displays the source code for 'diff3d.cc', with line 81 highlighted: `int rank = MPI::COMM_WORLD.Get_rank();`. The 'Locals' pane on the right shows the current line(s) and the variable 'rank' with a value of 32767. The bottom section contains tabs for 'Input/Output\*', 'Breakpoints', 'Watchpoints', 'Stacks', 'Tracepoints', and 'Tracepoint Output'. The 'Stacks' pane is active, showing a list of processes and threads, with the current stack frame being 'main (diff3d.cc:81)'. The 'Evaluate' pane is also visible, showing the expression and value.

Session Control Search View Help

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

All 0 1 2 3

Root 0

Workers 1 2 3

Create Group

Project Files Search (Ctrl+K)

diff3d.cc

```
74 }
75 // MPI::COMM_WORLD.Abort(1);
76 }
77
78 const int nthreads = get_num_threads();
79 const int root = 0;
80 const int size = MPI::COMM_WORLD.Get_size();
81 int rank = MPI::COMM_WORLD.Get_rank();
82
83 cerr << "nthreads=" << nthreads << endl;
84
85 //#include "mpidebug.ch"
86
87 mpiCommit<Parameters>();
88
```

Locals Current Line(s) Current Stack

Current Line(s)

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

Type: none selected

Input/Output\* Breakpoints Watchpoints Stacks Tracepoints Tracepoint Output Evaluate

Stacks

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

Ready

compute • calcul CANADA



# User interface (2)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. A blue callout box at the top center contains the text "DDT uses a tabbed-document interface." with arrows pointing to the "diff3d.cc" tab in the Project Files pane and the code editor window.

The interface includes a menu bar (Session Control, Search, View, Help), a toolbar, and a "Current Group" field. Below this is a process tree with nodes for "All", "Root", and "Workers", each with associated colored bars and numerical values.

The "Project Files" pane on the left shows a tree view of files, with "diff3d.cc" selected. The central code editor displays the following code:

```
74     }
75     // MPI::COMM_WORLD.Abort(1);
76     }
77
78     const int nthrds = get_num_threads();
79     const int root = 0;
80     const int size = MPI::COMM_WORLD.Get_size();
81     int rank = MPI::COMM_WORLD.Get_rank();
82
83     cerr << "nthrds=" << nthrds << endl;
84
85     // #include "mpidebug.ch"
86
87     mpiCommit<Parameters>();
88
```

The "Locals" pane on the right shows the current line(s) and variable values:

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

The "Stacks" pane at the bottom shows the current stack frames:

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

Other panes include "Input/Output\*", "Breakpoints", "Watchpoints", "Stacks", "Tracepoints", "Tracepoint Output", "Evaluate", and "Expression Value".

# User interface (3)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. A blue callout box with white text states: "When the session begins, DDT automatically finds source code from information compiled in the executable." An arrow points from this box to the code editor, which shows the following C code snippet:

```
74     }  
75     // MPI::COMM_WORLD.Abort(1);  
76     }  
77  
78     const int nthreads = get_num_threads();  
79     const int root = 0;  
80     const int size = MPI::COMM_WORLD.Get_size();  
81     int rank = MPI::COMM_WORLD.Get_rank();  
82  
83     cerr << "nthreads=" << nthreads << endl;  
84  
85     // #include "mpidebug.ch"  
86  
87     mpiCommit<Parameters>();  
88  
89
```

The Variable Name window shows the variable `MPI::COMM_...` with a value of `rank` and a value of `32767`.

The Stacks window shows the following stack frames:

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

# User interface (4)

The screenshot shows the Allinea DDT v3.1 (on gpc-f102n084) interface. At the top, there is a menu bar with 'Session', 'Control', 'Search', 'View', and 'Help'. Below the menu bar is a toolbar with various icons. A status bar indicates 'Current Group: All' and 'Focus on current: Group Process Thread Step Threads Together'. The main area displays a tree view of process groups: 'All' (blue bar with sub-groups 0, 1, 2, 3), 'Root' (green bar with sub-group 0), and 'Workers' (yellow bar with sub-groups 1, 2, 3). An arrow points from the 'Workers' group to a callout box. The callout box contains the following text:

*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, ...)

The background interface also shows a 'Project Files' list with 'diff3d.cc' selected, a 'Stacks' panel, and a 'Processes' panel showing threads for 'main (diff3d.cc:81)' and 'mxm\_event\_cleanup'.

# User interface (5)

Session Control Search View Help

Current Group: All

All

Root

Workers

Create Group

Project Files

Search (Ctrl+K)

diff3d.cc

```
74     }
75     // MPI::COMM_WORLD.Abort(1);
76     }
77
78     const int nthrds = get_num_threads();
79     const int root = 0;
80     const int size = MPI::COMM_WORLD.Get_size();
81     int rank = MPI::COMM_WORLD.Get_rank();
82
83     cerr << "nthrds=" << nthrds << endl;
84
85     //include "mpidebug.ch"
86
87     mpiCommit<Parameters>();
88
```

Locals Current Line(s) Current Stack

Current Line(s)

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

Type: none selected

Input/Output\* Breakpoints Watchpoints Stacks Tracepoints Tracepoint Output Evaluate

Stacks

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

Ready

compute • calcul  
CANADA

# User interface (6)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. At the top, there is a menu bar with 'Session', 'Control', 'Search', 'View', and 'Help'. Below the menu is a toolbar with various icons. A 'Current Group' dropdown is set to 'All', and 'Focus on current' has radio buttons for 'Group', 'Process', and 'Thread'. A 'Step Threads Together' checkbox is also present. Below this, a row of four colored buttons (0, 1, 2, 3) is visible. A blue callout box with a white border is overlaid on the interface, containing the text: *Session Control Dialog:* Control program execution, e.g., play/continue, pause, step into, step out. The main area is divided into several panes. On the left is a 'Project Files' tree view showing folders like 'del\_opv.cc', 'delete.c', 'diff3d.cc', 'distances.c', and 'divtf3.c'. The central pane shows a code editor with C++ code, including lines for thread counts and MPI rank. The right pane shows a 'Variable Name' table with 'rank' set to 32767. At the bottom, there are tabs for 'Input/Output\*', 'Breakpoints', 'Watchpoints', 'Stacks', 'Tracepoints', and 'Tracepoint Output'. The 'Stacks' tab is active, showing a table with columns for 'Processes', 'Threads', and 'Function'. The stack entries are: 4 processes, 4 threads, gomp\_thread\_start (team.c:120); 4 processes, 4 threads, main (diff3d.cc:81); and 4 processes, 4 threads, mxm\_event\_cleanup. The status bar at the bottom right shows 'Ready'.

Session Control Dialog:  
Control program execution, e.g., play/continue,  
pause, step into, step out

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 (7)

The screenshot displays the Allinea DDT v3.1 (on gpc-f102n084) interface. At the top, there is a menu bar with 'Session', 'Control', 'Search', 'View', and 'Help'. Below the menu is a toolbar with various icons. A 'Current Group' dropdown is set to 'All', and 'Focus on current' options for 'Group', 'Process', and 'Thread' are visible. A tree view on the left shows a hierarchy: 'All' (with sub-items 0, 1, 2, 3), 'Root' (with sub-item 0), and 'Workers' (with sub-items 1, 2, 3). Below this is a 'Create Group' button. The 'Project Files' pane shows 'diff3d.cc' as the active file. The 'Search (Ctrl+K)' pane is empty. The main editor window shows code from 'diff3d.cc' with lines 85, 86, and 87 visible. A blue callout box with a white border points to the 'Breakpoints' tab in the bottom pane, containing the text: *Breakpoints Tab*  
Can suspend, jump to, delete, load, save. The bottom pane also includes tabs for 'Input/Output\*', 'Watchpoints', 'Stacks', 'Tracepoints', and 'Tracepoint Output'. The 'Stacks' pane shows a table with columns 'Processes', 'Threads', and 'Function'. The 'Evaluate' pane is also visible.

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

# User interface (8)

The screenshot shows the Alinea DDT v3.1 (on gpc-f102n084) interface. At the top, the 'Focus on current' control is highlighted with a box, showing radio buttons for 'Group', 'Process', and 'Thread', with 'Group' selected. Below this, a row of four colored buttons (0, 1, 2, 3) is visible, with an arrow pointing to button '2'. A blue callout box with a white border contains the text: *Focus:* Choose between Group, process or thread. The interface also shows a file explorer on the left with 'diff3d.cc' selected, a code editor in the center with the following code snippet:

```
75 // MPI::COMM_WORLD.Abort(1);
76
77
78 const int nthreads = get_num_threads();
79 const int root = 0;
80 const int size = MPI::COMM_WORLD.Get_size();
81 int rank = MPI::COMM_WORLD.Get_rank();
82
83 cerr << "nthreads=" << nthreads << endl;
84
85 //#include "mpidebug.ch"
86
87 mpiCommit<Parameters>();
88
```

On the right, a 'Current Line(s)' panel shows a table with 'rank' having a value of 32767. At the bottom, the 'Stacks' panel shows a table with the following data:

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

# User interface (9)

The screenshot displays the Alinea DDT v3.1 interface. A blue callout box highlights the 'Stacks' feature. The interface includes a menu bar (Session, Control, Search, View, Help), a toolbar, and a main workspace with a file explorer on the left, a code editor in the center, and several panels at the bottom: Input/Output, Breakpoints, Watchpoints, Stacks, Tracepoints, Tracepoint Output, Evaluate, and a Stacks table.

**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

The 'Stacks' panel shows a call stack with the following entries:

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

The 'Evaluate' panel shows the variable `rank` with a value of `32767`.



# User interface (9)

The screenshot shows the Alinea DDT v3.1 (on gpc-f102n084) interface. A callout box with a blue border and white background contains the following text:

*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

The interface includes a menu bar (Session Control, Search, View, Help), a toolbar, and a main workspace. The workspace is divided into several panels:

- Current Group:** All
- Focus on current:** C Group, C Process, C Thread, Step, Threads Together
- Locals:** Current Line(s), Current Stack
- Current Line(s):** MPI::COMM\_... rank (Value: 32767)
- Input/Output\*, Breakpoints, Watchpoints, Stacks, Tracepoints, Tracepoint Output, Evaluate**
- Stacks:** Expression, Value

The **Stacks** panel is expanded, showing a table with the following data:

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

The **Locals** panel shows the variable `rank` with a value of `32767`. The **Current Line(s)** panel shows the source code for `diff3d.cc` at line 81:

```
82  cerr << "nthreads=" << nthreads << endl;  
83  
84  
85  // #include "mpidebug.ch"  
86  
87  mpiCommitParameters();  
88
```

# User interface (10)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. At the top, the 'Session Control Search View Help' menu is visible. Below it, the 'Current Group' is set to 'All'. A blue box highlights the 'Current line variables' panel, which shows the following data:

Group	0	1	2	3
All				
Root	0			
Workers	1	2	3	

The 'Project Files' window shows a list of files, with 'diff3d.cc' selected. The code editor displays the following code snippet:

```
74     }  
75     // MPI::COMM_WORLD.Abort(1);  
76     }  
77  
78     const int nthreads = get_num_threads();  
79     const int root = 0;  
80     const int size = MPI::COMM_WORLD.Get_size();  
81     int rank = MPI::COMM_WORLD.Get_rank();  
82  
83     cerr << "nthreads=" << nthreads << endl;  
84  
85     //include "mpidebug.ch"  
86  
87     mpiCommit<Parameters>();  
88
```

The 'Locals' window shows the current line variables:

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

The 'Stacks' window shows the current stack of frames:

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)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. At the top, the 'Session Control Search View Help' menu is visible. Below it, the 'Current Group: All' is shown. A blue box highlights the 'Local variables for process' section, which contains a grid of process group buttons: 'All' (0, 1, 2, 3), 'Root' (0), and 'Workers' (1, 2, 3). The 'Project Files' pane on the left shows a tree view with 'diff3d.cc' selected. The main editor window displays the source code for 'diff3d.cc', with line 81 highlighted: `int rank = MPI::COMM_WORLD.Get_rank();`. The 'Locals' pane on the right shows the current line(s) and the local variable 'rank' with a value of 32767. The bottom pane shows the 'Stacks' section with three entries: 'gomp\_thread\_start (team.c:120)', 'main (diff3d.cc:81)', and 'mxm\_event\_cleanup'. The 'Evaluate' pane is empty.

Local variables for process

```
74     }
75     // MPI::COMM_WORLD.Abort(1);
76 }
77
78 const int nthreads = get_num_threads();
79 const int root = 0;
80 const int size = MPI::COMM_WORLD.Get_size();
81 int rank = MPI::COMM_WORLD.Get_rank();
82
83 cerr << "nthreads=" << nthreads << endl;
84
85 //include "mpidebug.ch"
86
87 mpiCommit<Parameters>();
88
```

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 (12)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. At the top, there is a menu bar with 'Session Control Search View Help' and a toolbar with various icons. Below the toolbar, the 'Current Group' is set to 'All'. A blue box labeled 'Evaluate window' is overlaid on the top part of the interface, with an arrow pointing to the 'Evaluate' button in the bottom right corner.

The main area is divided into several panes:

- Process Group:** Shows 'All' (0, 1, 2, 3), 'Root' (0), and 'Workers' (1, 2, 3).
- Project Files:** A list of files including 'del\_opv.cc', 'del\_opvnt.cc', 'delete.c', 'diff3d.cc' (selected), 'distances.c', and 'divtf3.c'.
- Code Editor:** Displays the source code for 'diff3d.cc'. The current line is 81: `int rank = MPI::COMM_WORLD.Get_rank();`.
- Locals:** Shows the current line(s) and the value of the variable 'rank' as 32767.
- Stacks:** Shows the current stack frame, including 'gomp\_thread\_start (team.c:120)', 'main (diff3d.cc:81)' (selected), and 'mxm\_event\_cleanup'.

At the bottom, there is a 'Ready' status bar and a 'compute + calcul CANADA' logo.

## 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!

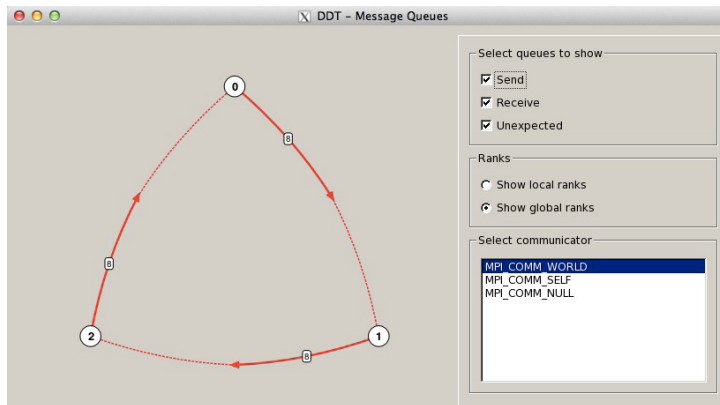
## Other features of DDT (2)

- ▶ 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



The screenshot displays the 'DDT - Message Queues' application window. On the left, a communication graph shows three nodes (0, 1, 2) arranged in a triangle. Solid red arrows indicate active communication paths between nodes, while dashed red lines represent potential or inactive paths. Each edge is labeled with the letter 'B'. On the right, there are three control panels:

- Select queues to show:** Contains three checked checkboxes:  Send,  Receive, and  Unexpected.
- Ranks:** Contains two radio buttons:  Show local ranks and  Show global ranks.
- Select communicator:** A list box containing three entries: `MPI_COMM_WORLD` (highlighted in blue), `MPI_COMM_SELF`, and `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



## DDT Hands-on...

# Useful references

- ▶ G Wilson  
*Software Carpentry* [software-carpentry.org/3\\_0/debugging.html](https://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](https://sources.redhat.com/gdb)
- ▶ *DDT*: [www.allinea.com/products/ddt-support](http://www.allinea.com/products/ddt-support)
- ▶ *SciNet Wiki*: [wiki.scinethpc.ca](http://wiki.scinethpc.ca): Tutorials & Manuals

## Example code 1

```
#include <stdlib.h>
void print_scrambled(char * msg);
int main() {
    char * bad_msg;
    bad_msg=NULL;
    char * good_msg="Hello world";
    print_scrambled(good_msg);
    print_scrambled(bad_msg);
    return 0;
}
```

```
#include <stdio.h>
void print_scrambled(char * msg) {
    int i=3;
    do {
        printf("%c", msg[i]);
        i++;
    } while (++msg);
    printf("\n");
}
```

## Example code 2

```
#include <stdio.h>
int main(int argc, char ** argv)
{
    unsigned int count = 999;
    unsigned int step = 2;
    do {
        printf("countdown at %d\n", count);
        count -= step;
    } while ( count >= 0 );
    printf("lift-off!\n");
    return 0;
}
```

## Example code 3

```
#include <mpi.h>
int main(int argc, char ** argv)
{
    MPI_Init(&argc,&argv);
    int procs, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int left = (rank+procs-1)%procs;
    int right = (rank+1)%procs;
    doubled = rank*1.1;
    MPI_Ssend(&d, 1, MPI_DOUBLE, right, 17, MPI_COMM_WORLD);
    MPI_Recv(&d, 1, MPI_DOUBLE, left, 17, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Finalize();
}
```

## Example code 4

```
#include <stdlib.h>
#include <stdio.h>
double* add_vector(double* a,double* b)
{
    double* c = malloc(3*sizeof(double));
    for (int i=0;i<3;i++)
        c[i] = a[i]+b[i];
    return c;
}
int main(int argc, char ** argv)
{
    double to_sum[10][3] = {
        {1,0,0}, {1,1,1}, {1,2,0}, {1,3,1}, {1,4,0},
        {2,0,1}, {2,1,0}, {2,2,1}, {2,3,0}, {2,4,1}
    };
    double answer[3] = {0,0,0};
    for (int i=0;i<10;i++)
        answer = add_vector(answer,to_sum[i]);
    printf("answer = %lf %lf %lf\n", answer[0], answer[1], answer[2]);
    return 0;
}
```

## Example code 5

```
#include <stdlib.h>
#include <stdio.h>
double ndot(int n, double*x, double*y){
    double tot=0;
    #pragma omp parallel for shared(x,y,n,tot)
    for (int i=0; i<n; i++)
        tot += x[i] * y[i];
    return tot;
}
int main(int argc, char** argv){
    int n = 10000000;
    double*x = malloc(n*sizeof(double));
    double*y = malloc(n*sizeof(double));
    for (int i=0; i<n; i++) {
        x[i] = i;
        y[i] = i;
    }
    double nn=n-1;
    double ans=nn*(nn+1)*(2*nn+1)/6.0;
    double dot=ndot(n,x,y);
    return 0;
}
```