

Debugging

Ramses van Zon
SciNet HPC Consortium
University of Toronto

July 17, 2015



Outline

- ▶ Debugging Basics
- ▶ Debugging with the command line: GDB
- ▶ Memory debugging with the command line: valgrind
- ▶ (Parallel) 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 (<code>sqrt(-0.0)</code>), 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.**

How to avoid debugging:

- ▶ Write better code.
 - ▶ Simpler, clear, straightforward code.
 - ▶ Modularity (no global variables or 10,000-line functions)
 - ▶ Avoid 'cute' tricks (no obfuscated C code winners)
- ▶ Don't write code, use existing libraries
- ▶ Write (simple) tests for each part of your code
- ▶ Use version control so you can 'roll back'.

Debugging Workflow

First things first:

- ▶ As soon as you are convinced there is a real problem, create the simplest situation in which it reproducibly occurs.
- ▶ This is science: model, hypothesis, experiment, conclusion.
- ▶ Try a smaller problem size, turning off physical effects with options, etc. until you have a simple, fast repeatable example of the bug.
- ▶ Try to narrow it down to a particular module/function/class. For fortran, switch on bounds checking (**-fbounds-check.**)
- ▶ Now you're ready to start debugging.

Ways to debug

Ways to debug

- ▶ Preemptive:
 - ▶ Turn on compiler warnings: fix or understand them!
`$ gcc/gfortran -Wall`
 - ▶ Check your assumptions (e.g. use **assert**).

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!

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

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

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

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?*

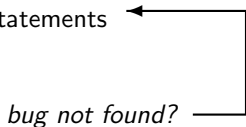
What's wrong with using print statements?

Strategy

► Constant cycle:

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

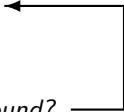
bug not found?



What's wrong with using print statements?

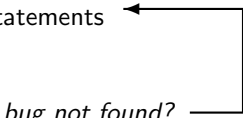
Strategy

- ▶ 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

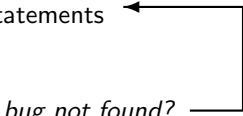
What's wrong with using print statements?

Strategy

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

What's wrong with using print statements?

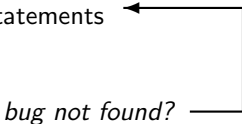
Strategy

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

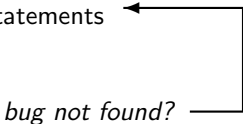
- ▶ Constant cycle:
 1. strategically add print statements
 2. compile
 3. run
 4. analyze output
 - ▶ Removing the extra code after the bug is fixed
 - ▶ Repeat for each bug
- 
- bug not found?*

Problems with this approach

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

What's wrong with using print statements?

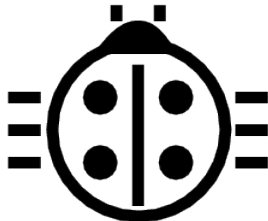
Strategy

- ▶ Constant cycle:
 1. strategically add print statements
 2. compile
 3. run
 4. analyze output
 - ▶ 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... **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): can be slow

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

Symbolic debuggers

Preparing the executable

- ▶ Add required compilation flags:
 - \$ gcc/g++/gfortran -g [-gstabs]
 - \$ icc/icpc/ifort -g [-debug parallel]
 - \$ nvcc -g -G
- ▶ Optional: switch off optimization -O0

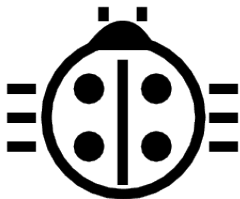
Symbolic debuggers

Preparing the executable

- ▶ Add required compilation 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/4.7.2
$ gcc -Wall -g -O0 example.c -o example
$ module load gdb/7.6
$ gdb -tui example
...
(gdb)_
```

GDB basic building blocks



Demonstration of GDB features

- ▶ We will look at the features of gdb using a running example.
- ▶ Example reads integers from command line and sums them.
- ▶ There's a C and a Fortran version.

```
$ ssh USER@login.scinet.utoronto.ca -X
$ ssh gpc01 -X
$ qsub -l nodes=1:ppn=8,walltime=8:00:00 -I -X -qteach
$ cp -r /scinet/course/ss2015/debug $SCRATCH
$ source $SCRATCH/debug/code/setup
$ cd $SCRATCH/debug/code/bugexample
$ make bugexample #(or make bugexample_f)
```

GDB building block #1: Inspect crashes

Inspecting core files

GDB building block #1: Inspect crashes

Inspecting core files

Core = file containing state of program after a crash

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

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?

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.

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.

Related gdb commands

run	run the executable from the start
list	list code lines (where current execution is, or range)

GDB building block #1: Inspect crashes

```
$ ulimit -c 1024
```

```
$ ./bugexample #(or ./bugexample_f)
```

Give some integers as command-line arguments

```
$ ./bugexample 1 3 5
```

```
Segmentation fault (core dumped)
```

GDB building block #1: Inspect crashes

```
$ ulimit -c 1024
```

```
$ ./bugexample #(or ./bugexample_f)
```

Give some integers as command-line arguments

```
$ ./bugexample 1 3 5
```

```
Segmentation fault (core dumped)
```

```
$ gdb ./bugexample core.2387 # core number varies
```

GDB building block #1: Inspect crashes

```
$ ulimit -c 1024
$ ./bugexample #(or ./bugexample_f)
Give some integers as command-line arguments
$ ./bugexample 1 3 5
Segmentation fault (core dumped)

$ gdb ./bugexample core.2387 # core number varies
```

GNU gdb (GDB) 7.6

Copyright (C) 2013 Free Software Foundation, Inc.

...

```
Reading symbols from debug/code/bugexample/bugexample...done.
[New LWP 3817]
```

```
warning: Can't read pathname for load map: Input/output error.
Core was generated by './bugexample 1 3 5'.
```

```
Program terminated with signal 11, Segmentation fault.
```

```
#0 0x4007d5 in sum_integers (n=3, a=0x4) at intltools.c:30
```

```
30      s += a[i];
```

```
(gdb)
```

GDB building block #1: Inspect crashes

...

Program terminated with signal 11, Segmentation fault.

```
#0 0x4007d5 in sum_integers (n=3, a=0x4) at intlisttools.c:30
```

```
30          s += a[i];
```

This points at the line where the error is detected.

GDB building block #1: Inspect crashes

...

Program terminated with signal 11, Segmentation fault.

```
#0 0x4007d5 in sum_integers (n=3, a=0x4) at intlittestools.c:30
30          s += a[i];
```

This points at the line where the error is detected. More context:

GDB building block #1: Inspect crashes

...

Program terminated with signal 11, Segmentation fault.

```
#0 0x4007d5 in sum_integers (n=3, a=0x4) at intlittestools.c:30
30          s += a[i];
```

This points at the line where the error is detected. More context:

```
(gdb) list
```

GDB building block #1: Inspect crashes

...

Program terminated with signal 11, Segmentation fault.

```
#0 0x4007d5 in sum_integers (n=3, a=0x4) at intlisttools.c:30
30          s += a[i];
```

This points at the line where the error is detected. More context:

```
(gdb) list
```

```
25 /* Compute the sum of the array of integers */
26 int sum_integers(int n, int* a)
27 {
28     int i, s;
29     for (i=0; i<n; i++)
30         s += a[i];
31     return s;
32 }
```

```
(gdb)
```


GDB building block #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 block #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 block #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 block #2: Function call stack

...

```
(gdb) list
```

```
25 /* Compute the sum of the array of integers */
```

```
26 int sum_integers(int n, int* a)
```

```
27 {
```

```
28     int i, s;
```

```
29     for (i=0; i<n; i++)
```

```
30         s += a[i];
```

```
31     return s;
```

```
32 }
```

```
(gdb) backtrace
```

```
#0 0x4007d5 in sum_integers (n=3,a=0x4) at intlittestools.c:30
```

```
#1 0x40082a in process (argc=4,argv=0x7fff0b89ce58) at process.c
```

```
#2 0x4006d3 in main (argc=4,argv=0x7fff0b89ce58) at bugexample.c
```

```
(gdb)
```

GDB building block #3: 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 block #3: 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>	print variable
<code>display</code>	print at every prompt
<code>set variable</code>	change variable
<code>watch</code>	stop if variable changes

GDB building block #3: Variables

Remember: We were looking at a seg fault in `s += a[i]`.

```
(gdb) print i
```

```
0
```

```
(gdb) print a[0]
```

```
Cannot access memory at address 0x4
```

```
(gdb) print a
```

```
0x4
```

```
(gdb) up
```

```
#1 0x00000000040082a in process (argc=4, argv=0x7fff0b89ce58)
```

```
11 int s = sum_integers(n, arg);
```

```
(gdb) print arg
```

```
$1 = (int *) 0x4
```

```
(gdb) list
```

```
7 void process(int argc, char** argv)
```

```
8 {
```

```
9     int* arg = read_integer_arguments(argc, argv);
```

```
10     int n = argc-1;
```

```
11     int s = sum_integers(n, arg);
```

```
12     print_integers(n, arg);
```

```
13     printf("Sum of integers is: %d\n", s);
```

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.

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 #4: Automatic interruption

```
...
(gdb) list
7 void process(int argc, char** argv)
8 {
9     int* arg = read_integer_arguments(argc, argv);
10    int n = argc-1;
11    int s = sum_integers(n, arg);
12    print_integers(n, arg);
13    printf("Sum of integers is: %d\n", s);
14    free(arg);
15 }
(gdb) break read_integer_arguments
Breakpoint 1 at 0x4006ec: file intlisttools.c, line 8.
```

GDB building block #4: Automatic interruption

```
...
(gdb) list
7 void process(int argc, char** argv)
8 {
9     int* arg = read_integer_arguments(argc, argv);
10    int n = argc-1;
11    int s = sum_integers(n, arg);
12    print_integers(n, arg);
13    printf("Sum of integers is: %d\n", s);
14    free(arg);
15 }
(gdb) break read_integer_arguments
Breakpoint 1 at 0x4006ec: file intlisttools.c, line 8.

(gdb) run 1 3 5
```

GDB building block #4: Automatic interruption

...

```
(gdb) list
```

```
7 void process(int argc, char** argv)
8 {
9     int* arg = read_integer_arguments(argc, argv);
10    int n = argc-1;
11    int s = sum_integers(n, arg);
12    print_integers(n, arg);
13    printf("Sum of integers is: %d\n", s);
14    free(arg);
15 }
```

```
(gdb) break read_integer_arguments
```

```
Breakpoint 1 at 0x4006ec: file intlisttools.c, line 8.
```

```
(gdb) run 1 3 5
```

```
Starting program: debug/code/bugexample/bugexample 1 3 5
```

```
Breakpoint 1, read_integer_arguments (n=4, a=0x7fffffff9b8)
at intlisttools.c:8
```

```
8     int* result = malloc(sizeof(int)*(n-1));
```

```
(gdb)
```

GDB building block #5: 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 block #5: 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 #5: Step through code

...

```
(gdb) list 6,14
```

```
6 int* read_integer_arguments(int n, char** a)
```

```
7 {
```

```
8     int* result = malloc(sizeof(int)*(n-1));
```

```
9     int i;
```

```
10     /* convert every argument, but skip '0', because it is ju
```

```
11         executable name */
```

```
12     for (i=1;i<n;i++)
```

```
13         result[i] = atoi(a[i]);
```

```
14 }
```

```
(gdb) display result
```

```
1: result = (int *) 0x0
```

```
(gdb) next
```

```
12     for (i=1;i<n;i++)
```

```
1: result = (int *) 0x601010
```

```
(gdb) until 14
```

GDB building block #5: Step through code

```
(gdb) until 14
read_integer_arguments (n=4,a=0x7fffffff9b8) at intlisttools.c:
14 }
1: result = (int *) 0x601010
(gdb) finish
Run till exit from #0  read_integer_arguments (n=4,
      a=0x7fffffff9b8) at intlisttools.c:14
0x00000000040080c in process (argc=4, argv=0x7fffffff9b8)
      at process.c:9
9      int* arg = read_integer_arguments(argc, argv);
Value returned is $3 = (int *) 0x4
(gdb)
```


GDB building block #5: Step through code

```
(gdb) until 14
read_integer_arguments (n=4,a=0x7fffffff9b8) at intlisttools.c:
14 }
1: result = (int *) 0x601010
(gdb) finish
Run till exit from #0  read_integer_arguments (n=4,
      a=0x7fffffff9b8) at intlisttools.c:14
0x00000000040080c in process (argc=4, argv=0x7fffffff9b8)
      at process.c:9
9      int* arg = read_integer_arguments(argc, argv);
Value returned is $3 = (int *) 0x4
(gdb)
```

He, why is the result variable equal to **0x601010** while the value returned is **0x4**?

GDB building block #5: Step through code

```
(gdb) until 14
read_integer_arguments (n=4,a=0x7fffffff9b8) at intlisttools.c:
14 }
1: result = (int *) 0x601010
(gdb) finish
Run till exit from #0  read_integer_arguments (n=4,
      a=0x7fffffff9b8) at intlisttools.c:14
0x00000000040080c in process (argc=4, argv=0x7fffffff9b8)
      at process.c:9
9      int* arg = read_integer_arguments(argc, argv);
Value returned is $3 = (int *) 0x4
(gdb)
```

He, why is the result variable equal to **0x601010** while the value returned is **0x4**?

Contradicts your assumption of what the program does.

The program is always right, you are wrong.

GDB building block #5: Step through code

Why is the result variable equal to `0x601010` while the value returned is `0x4`?

GDB building block #5: Step through code

Why is the result variable equal to **0x601010** while the value returned is **0x4**?

```
(gdb) list read_integer_arguments,+7
7 {
8     int* result = malloc(sizeof(int)*(n-1));
9     int i;
10    /* convert every argument, but skip '0', because it is ju
11       executable name */
12    for (i=1;i<n;i++)
13        result[i] = atoi(a[i]);
14 }
```

GDB building block #5: Step through code

Why is the result variable equal to **0x601010** while the value returned is **0x4**?

```
(gdb) list read_integer_arguments,+7
7 {
8     int* result = malloc(sizeof(int)*(n-1));
9     int i;
10    /* convert every argument, but skip '0', because it is ju
11       executable name */
12    for (i=1;i<n;i++)
13        result[i] = atoi(a[i]);
14 }
```

Aargh! Forgot the return statement!

GDB building block #5: Step through code

Why is the result variable equal to **0x601010** while the value returned is **0x4**?

```
(gdb) list read_integer_arguments,+7
7 {
8     int* result = malloc(sizeof(int)*(n-1));
9     int i;
10    /* convert every argument, but skip '0', because it is ju
11        executable name */
12    for (i=1;i<n;i++)
13        result[i] = atoi(a[i]);
14 }
```

Aargh! Forgot the return statement!

Feeling like an idiot is a common side-effect of debugging.

GDB command summary

help	h	print description of
run	r	run from the start (+args)
backtrace/where	ba	function call stack
list	l	list code lines
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
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
quit	q	quit gdb

Memory Debugging



Memory Checking: Valgrind

- ▶ Memory errors do not always give segfaults
- ▶ Commonly have to go *way* out of bounds to get a segfault.
- ▶ Write into other variable - hard to find problem.
- ▶ **Valgrind** - intercepts each memory call and checks them.
- ▶ Finds illegal accesses, uninitialized values, memory leaks.
- ▶ Warning: Quite verbose, typically, and, if you use external libraries, sometimes false positives. debugging too.



Valgrind example

```
$ valgrind ./bugexample 1 3 5
==909== Memcheck, a memory error detector
==909== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward
==909== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyr
==909== Command: ./bugexample 1 3 5
==909==
==909== Invalid write of size 4
==909==    at 0x400741: read_integer_arguments (intltools.c:1
==909==    by 0x40080B: process (process.c:9)
==909==    by 0x4006D2: main (bugexample.c:12)
==909== Address 0x51c304c is 0 bytes after a block of size 12 a
==909==    at 0x4C2636D: malloc (vg_replace_malloc.c:291)
==909==    by 0x4006FF: read_integer_arguments (intltools.c:8
==909==    by 0x40080B: process (process.c:9)
==909==    by 0x4006D2: main (bugexample.c:12)
==909==
==909== Invalid read of size 4
...
```

Valgrind example (continued)

```
==909== HEAP SUMMARY:
==909==      in use at exit: 12 bytes in 1 blocks
==909==    total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==909==
==909== LEAK SUMMARY:
==909==    definitely lost: 12 bytes in 1 blocks
==909==    indirectly lost: 0 bytes in 0 blocks
==909==    possibly lost: 0 bytes in 0 blocks
==909==    still reachable: 0 bytes in 0 blocks
==909==    suppressed: 0 bytes in 0 blocks
==909== Rerun with --leak-check=full to see details of leaked memory
==909==
==909== For counts of detected and suppressed errors, rerun with
==909== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 6 from 6)
Segmentation fault
$ _
```

Valgrind recommendations

- ▶ Using valgrind on mature codes often shows lots of errors. Now, some may not be an issue (e.g. dead code or false positives from libraries), but hard to know.
- ▶ So: start using valgrind early in development.
- ▶ Program modularly, and create small unit tests, on which you can comfortably use valgrind.
- ▶ Apart from this basic valgrind usage, there are other tools available with valgrind to deal cache performance, to get more detailed memory leak information, to detect race conditions, etc. (some of which we'll discuss later).

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)

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

The screenshot displays the DDD graphical debugger interface. The main window shows a C program with OpenMP parallelism. A breakpoint is set at the start of the parallel region. The threads window shows four threads, with thread 3 selected. The control panel on the right includes buttons for Run, Interrupt, Step, Next, Until, Cont, Up, Undo, Edit, Step!, Next!, Finish, Kill, Down, Redo, and Make.

```
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);
}
```

Threads

- 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=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) |

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

Graphical symbolic debuggers - ddt

The screenshot displays the Alinea DDT v3.1 (on gcc-f102n084) graphical user interface. The main window shows the source code of `dm3d.cc` at line 105, where a `cout` statement is being executed. The code defines a point `p` with coordinates `dx`, `dy`, and `dz`, and prints them. The `Stacks` panel at the bottom left shows the current stack frame for the `main` function. The `Locals` panel on the right shows the current state of local variables, including `argc`, `argv`, `comm`, `coords`, `dms`, `field`, `fulnm`, `ini`, `lastt`, `negProc`, `negSlabOut`, `npoints`, `infrds`, `oldprogress`, `origin`, `p`, and `periods`. The `Evaluate` panel at the bottom right shows the expression `<no symbol 'r' in current context>`.

```
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.[0] = ini.get_double("x", 10);
99 p.[1] = ini.get_double("y", 10);
100 p.[2] = ini.get_double("z", 10);
101 p.r[0] = ini.get_long("nx", 10);
102 p.r[1] = ini.get_long("ny", 10);
103 p.r[2] = ini.get_long("nz", 10);
104
105 cout << "r = "
106 << p.[0] << " "
107 << p.[1] << " "
108 << p.[2] << "\n"
109 << "n = "
110 << p.r[0] << " "
111 << p.r[1] << " "
112 << p.r[2] << "\n";
113
114 // points per processor
115 double pop = (p.r[0]*p.r[1]*p.r[2])/size;
116 n.drm[0] = n.drm[1] = n.drm[2] = 1;
```

Processes	Threads	Function
1	1	~_kmp_launch_monitor
1	1	~_kmp_launch_worker
1	1	~bbi_openib_async_thread
1	1	main (dm3d.cc:105)
1	1	~service_thread_start

Variable Name	Value
-argc	2
-argv	0x7fffffc
-comm	
-coords	
-dms	0x17
-field	0x7ffffe2
-fulnm	
-ini	
-lastt	14073729
-negProc	
-negSlabOut	
-npoints	14073735
-infrds	2
-oldprogress	1342464
-origin	
-p	
-periods	

Expression	Value
<no symbol 'r' in current context>	

DDT



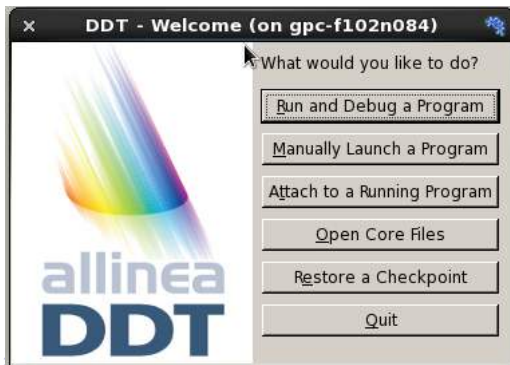
- ▶ “Distributed Debugging Tool”
- ▶ Powerful GUI-based commercial debugger by *Allinea*.
- ▶ Supports C, C++ and Fortran
- ▶ Supports MPI, OpenMP, threads, CUDA and more
- ▶ Available on all SciNet clusters (GPC, TCS, ARC, P7)
- ▶ Available on SHARCNET’s kraken, requin, orca and monk.
- ▶ Part of the “Allinea Forge” suite, which also includes a ‘profiler’ called *MAP*.

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 two overlapping dialog boxes from the DDT (Data Display Tool) interface. The background dialog is titled "DDT - Run (on gpc-f102n084)" and contains configuration options for running a program. The foreground dialog 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%

Run and Debug a Program (session setup)

DDT - Run (on gpc-f102n084)

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 ^

Run

Cancel

Memory Debugging Options (o

Preload the memory debugging library: Lang

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

Heap Debugging

- 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: 1 Add guard pages: Aft

Advanced

- Specify heap-check interval:** 100
- Store stack backtraces for memory allocations**
- Only enable for these processes:**

0-1

100%

Select

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 the DDT (Data Display Tool) interface for running a program. The main window is titled "DDT - Run (on gpc-f102n084)". It contains several sections for configuring the execution environment:

- 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:** (unchecked).
- Memory Debugging:** Minimal, No guard pages, Backtraces, Preload.
- Environment Variables:** none.
- Plugins:** none.

The "Memory Debugging Options (on gpc-f102n084)" sub-dialog box is open, showing the following settings:

- 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 for "Run" and "Cancel" are visible at the bottom of the main dialog, and "OK" and "Cancel" are visible at the bottom of the sub-dialog.

User interface (1)

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 status bar shows 'Current Group: All' and 'Focus on current: Group Process Thread Step Threads Together'. The main area is divided into several panes:

- Group View:** Shows a tree view of the application's execution groups. 'All' is selected and contains four sub-groups: 'Root' (rank 0) and 'Workers' (ranks 1, 2, 3).
- Project Files:** A file explorer on the left showing a directory structure with files like 'del_opv.cc', 'diff3d.cc', etc.
- Code Editor:** Displays the source code for 'diff3d.cc'. Line 81 is highlighted, showing `int rank = MPI::COMM_WORLD.Get_rank();`.
- Locals:** A table showing local variables for the current line. The variable 'rank' has a value of 32767.
- Stacks:** A table showing the call stack. The current frame is 'main (diff3d.cc:81)', with other frames like 'gomp_thread_start' and 'mxm_event_cleanup' visible.
- Input/Output, Breakpoints, Watchpoints, Stacks, Tracepoints, Tracepoint Output:** A row of tabs for different debugging features.
- Evaluate:** A table for evaluating expressions, currently empty.

The bottom right corner shows the system status 'Ready' and the 'compute • calcul CANADA' logo.

User interface (2)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) user 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 window and the code editor window.

Project Files: A tree view on the left shows a project structure with files like `del_opv.cc`, `del_opvnt.cc`, `delete.c`, `diff3d.cc` (selected), `distances.c`, and `divtf3.c`.

Code Editor: The central window shows the source code for `diff3d.cc`. The current line is 81, which is `int rank = MPI::COMM_WORLD.Get_rank();`. The code includes MPI-related functions and thread management.

Stacks: The bottom window shows the current stack with three entries:

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

Current Line(s) Table: A table on the right shows the current line(s) and their values:

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

User interface (3)

The screenshot shows the Alinea DDT v3.1 (on gpc-f102n084) interface. A callout box points to line 81 in the `diff3d.cc` file, which contains the line `int rank = MPI::COMM_WORLD.Get_rank();`. The stack window shows the current function is `main (diff3d.cc:81)` and the variable `rank` has a value of `32767`.

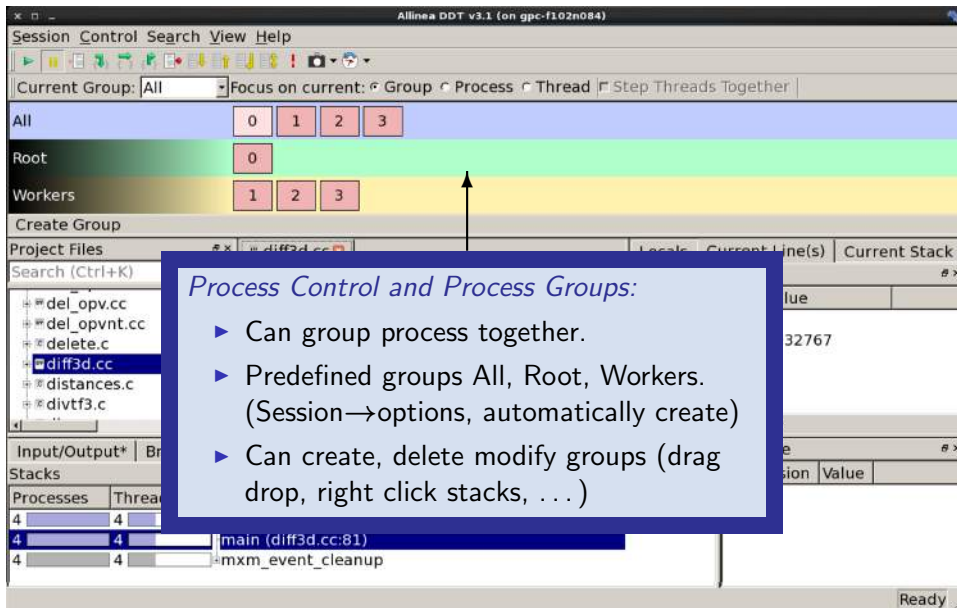
When the session begins, DDT automatically finds source code from information compiled in the executable.

```
74 | |
75 | // MPI::COMM_WORLD.Get_size();
76 | |
77 | |
78 | const int nthreads = get_nthreads();
79 | const int root = 0;
80 | const int nsize = MPI::COMM_WORLD.Get_size();
81 | int rank = MPI::COMM_WORLD.Get_rank();
82 | |
83 | cerr << "nthreads=" << nthreads << endl;
84 | |
85 | //include "spidebug.ch"
86 | |
87 | mpiCommitParameters();
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 (4)



The screenshot shows the Allinea DDT v3.1 interface. At the top, there's a menu bar with 'Session', 'Control', 'Search', 'View', and 'Help'. Below it is a toolbar with various icons. A control bar shows 'Current Group: All' and radio buttons for 'Focus on current: Group', 'Process', and 'Thread', along with a checkbox for '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 call stack window. The call stack shows 'main (diff3d.cc:81)' and 'mxm_event_cleanup'. A blue callout box is overlaid on the interface.

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)

The screenshot displays the Allinea DDT v3.1 (on gpc-f102n084) interface. A callout box with a blue border and white background contains the text: "Different colour coding for each group's current source line." An arrow points from this box to line 81 of the code in the editor, which is highlighted in blue. The code editor shows the following lines:

```
74 | | MPI::COMM_WORLD, dest ());  
75 | //  
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 "apidebug.ch"  
86 | |  
87 | mpiCommitParameters();  
88 | |
```

The interface also shows a "Project Files" pane on the left with a search bar and a list of files including `del_opv.cc`, `del_opvnt.cc`, `delete.c`, `diff3d.cc` (selected), `distances.c`, and `divtf3.c`. On the right, there are panes for "Locals", "Current Line(s)", and "Current Stack". The "Current Line(s)" pane shows a table with the following data:

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

At the bottom, there are panes for "Input/Output*", "Breakpoints", "Watchpoints", "Stacks", "Tracepoints", and "Tracepoint Output". The "Stacks" pane 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 (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, including a play button. A text box indicates 'Current Group: All' and 'Focus on current: Group Process Thread Step Threads Together'. A row of four buttons labeled '0', '1', '2', and '3' is visible. A blue callout box with the text 'Session Control Dialog: Control program execution, e.g., play/continue, pause, step into, step over, step out' points to the toolbar. The main area is divided into several panes: a file browser on the left showing files like 'del_opv.cc', 'diff3d.cc', and 'divtf3.c'; a central code editor showing C code with a line 'int rank = MPI::COMM_WORLD.Get_rank();' highlighted; a variable viewer on the right showing 'rank' with value '32767'; and a bottom pane with tabs for 'Input/Output*', 'Breakpoints', 'Watchpoints', 'Stacks', 'Tracepoints', and 'Tracepoint Output'. The 'Stacks' tab is active, showing a list of processes and threads, with 'main (diff3d.cc:81)' selected.

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

Process	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 status bar shows 'Current Group: All' and 'Focus on current: Group Process Thread Step Threads Together'. The main area is divided into several panes. The top pane shows a tree view of groups: 'All' (0, 1, 2, 3), 'Root' (0), and 'Workers' (1, 2, 3). Below this is a 'Project Files' pane showing 'diff3d.cc'. A 'Search (Ctrl+K)' pane is also visible. The 'Breakpoints Tab' is highlighted in blue, and a callout box points to it with the text: 'Breakpoints Tab Can suspend, jump to, delete, load, save'. The bottom pane shows a table of processes and threads:

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 interface. At the top, the 'Focus on current' dropdown menu is open, showing options for 'Group', 'Process', and 'Thread'. A blue callout box with a white border contains the text: *Focus:* Choose between Group, process or thread. An arrow points from the callout box to the 'Group' option in the dropdown menu. The interface also displays a project tree on the left, a code editor in the center, and a variable table on the right.

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

Focus:
Choose between Group, process or thread

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 (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

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. The top menu bar includes Session, Control, Search, View, and Help. Below the menu is a toolbar with various icons. The main window is divided into several panels:

- Current Group:** All
- Focus on current:** C Group, C Process, C Thread, C Step, Threads Together
- Stacks:** A tree of functions (merged) showing a hierarchy of processes and threads. The current stack is highlighted in blue.
- Current Stack:** A table showing the current stack frame. The variable `rank` has a value of `32767`.
- Locals:** A table showing local variables.
- Current Line(s):** A table showing the current line of code.
- Input/Output*, Breakpoints, Watchpoints, Tracepoints, Tracepoint Output:** Panels for debugging and monitoring.
- Evaluate:** A panel for evaluating expressions.

The Stacks panel shows 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 Current Stack panel shows the following data:

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

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

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. A blue callout box highlights the 'Stacks' panel, which is currently active. The 'Stacks' panel shows a tree of functions (merged) with three entries: 'gomp_thread_start (team.c:120)', 'main (diff3d.cc:81)', and 'mxm_event_cleanup'. The 'main' entry is selected. The 'Current Stack' panel shows the current line(s) and variable values, including 'rank' with a value of 32767. The 'Locals' panel is also visible, showing the current line(s) and variable values. The 'Input/Output*' panel is at the bottom, showing 'Breakpoints', 'Watchpoints', 'Stacks', 'Tracepoints', and 'Tracepoint Output'.

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

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

User interface (10)

The screenshot displays the Alinea DDT v3.1 (on gpc-f102n084) interface. A blue box highlights the 'Current line variables' section, which is currently empty. An arrow points from this box to the 'Current Line(s)' panel on the right, which shows the variable 'rank' with a value of 32767. The main editor window shows the source code for 'diff3d.cc' with line 81 highlighted. The 'Stacks' panel at the bottom shows the current function is 'main (diff3d.cc:81)'. The 'Processes' and 'Threads' panels show 4 processes and 4 threads.

Current line variables

```
74 | |
75 | // MPI::COMM_WORLD.Abort();
76 | |
77 | |
78 | const int nthreads = get_num_threads();
79 | const int root = 0;
80 | const int nsize = MPI::COMM_WORLD.Get_size();
81 | int rank = MPI::COMM_WORLD.Get_rank();
82 | |
83 | cerr << "nthreads=" << nthreads << endl;
84 | |
85 | //!include "apidebug.ch"
86 | |
87 | mpiCommitParameters();
88 | |
```

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

Process	Thread	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, there is a menu bar with 'Session', 'Control', 'Search', 'View', and 'Help'. Below the menu bar, a 'Current Group:' dropdown is set to 'All'. A blue callout box with the text 'Local variables for process' points to the 'Workers' section, which shows three worker processes (1, 2, 3) under the 'Root' process (0). The 'Project Files' pane on the left shows a tree view of files, 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, which is currently empty. The status bar at the bottom right indicates 'Ready'.

Local variables for process

```
74 | |
75 | // MPI::COMM_WORLD.Abort();
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 "apidebug.ch"
86 | |
87 | mpiCommitParameters();
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', and 'Help'. Below the menu bar is a toolbar with various icons. A blue box labeled 'Evaluate window' is positioned over the toolbar and the top part of the main workspace. The main workspace is divided into several panes: 'Current Group' (All), 'Project Files' (diff3d.cc), 'Code Editor' (diff3d.cc), 'Locals' (Current Line(s), Current Stack), 'Input/Output*', 'Breakpoints', 'Watchpoints', 'Stacks', 'Tracepoints', 'Tracepoint Output', and 'Evaluate'. The 'Evaluate' pane is active, showing a table with 'Variable Name' and 'Value' columns. The variable 'rank' has a value of 32767. The 'Stacks' pane shows the current stack frame: 'main (diff3d.cc:81)'. The 'Processes' and 'Threads' panes show 4 processes and 4 threads, with the function 'gomp_thread_start (team.c:120)' and 'mxm_event_cleanup' visible.

Session Control Search View Help

Current Group: All

0 1 2 3

Root 0

Workers 1 2 3

Create Group

Project Files

Search (Ctrl+K)

del_opv.cc

del_opvnt.cc

delete.c

diff3d.cc

distances.c

divtf3.c

diff3d.cc

```
74 | |
75 | // MPI::COMM_WORLD.Abort();
76 | |
77 | |
78 | const int nthreads = get_num_threads();
79 | const int root = 0;
80 | const int nsize = MPI::COMM_WORLD.Get_size();
81 | int rank = MPI::COMM_WORLD.Get_rank();
82 | |
83 | cerr << "nthreads=" << nthreads << endl;
84 | |
85 | // #include "apidebug.ch"
86 | |
87 | mpiCommitParameters();
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

First Demonstration DDT

```
$ cd $SCRATCH/debug/code  
$ source setup  
$ cd bugexample  
$ make  
$ ddt bugexample
```

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)

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 Memory Debugging with DDT

```
$ cd $SCRATCH/debug/code
```

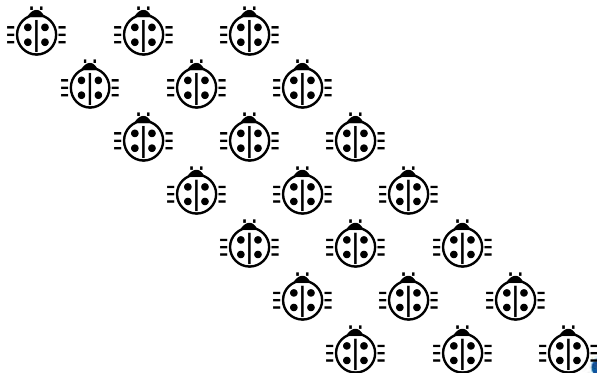
```
$ source setup
```

```
$ cd ex4
```

```
$ make
```

```
$ ddt ex4
```

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>`

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)

Shared memory debugging with DDT

Or use DDT:

Thread debugging example

```
$ cd $SCRATCH/debug/code  
$ source setup  
$ cd ex5  
$ make  
$ ddt ex5
```

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 demands specialized tools: ddt

Demonstration MPI Debugging with DDT

```
$ cd $SCRATCH/debug/code
```

```
$ source setup
```

```
$ cd ex2
```

```
$ make
```

```
$ ddt ex2
```

Detecting deadlock with DDT

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

Demonstration MPI Message Queue in DDT

```
$ cd $SCRATCH/debug/code
```

```
$ source setup
```

```
$ cd ex3
```

```
$ make
```

```
$ ddt ex3
```


Useful references

- ▶ N Matloff and PJ Salzman
The Art of Debugging with GDB, DDD and Eclipse
- ▶ *GDB*: sources.redhat.com/gdb
- ▶ *DDT*: www.allinea.com/knowledge-center/tutorials
- ▶ *SciNet Wiki*: wiki.scinethpc.ca: Tutorials & Manuals