

Debuggers and Parallel Debugging

HPC Best Practices

SciNet, Toronto

Debugging basics



Debugging basics

Help, my program doesn't work!

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

a miracle occurs

My program works brilliantly!

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

- Unfortunately, “miracles” are not yet supported by SciNet.

Debugging:

Methodical process of finding and fixing flaws in software

Debugging basics

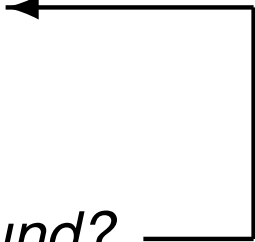
Ways to debug

- Don't write buggy code. **YEAH, RIGHT.**
- Add print statements **NO WAY TO DEBUG!**
- Command-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](#)
Excellent for parallel debugging, and available at SciNet!

What's wrong with using print statements?

Print debugging

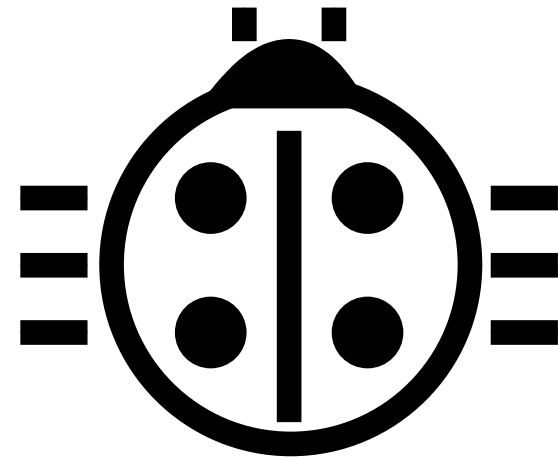
- Constant cycle:
 1. strategically add print statements
 2. compile
 3. run
 4. analyze output

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

Problems

- Time consuming
- Error prone
- Changes memory, timing... **THERE'S A BETTER WAY!**

Symbolic debuggers



Symbolic debuggers

Features

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

Use a graphical debugger or not?

- Local work station: graphical is convenient
- Remotely (SciNet):
 - Graphical debuggers slow
 - Graphics may not be available
 - Command-based debuggers fast (esp. gdb).
- Graphical debuggers still have command prompt.

Symbolic debuggers

Preparing the executable

- Required: compile with `-g`.
- Optional: switch off optimization `-O0`

Command-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 building blocks



GDB basics - 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 basics - 2 Function call stack

Interrupting program

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

Stack trace

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

gdb commands

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

GDB basics - 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 basics - 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  
condition  
disable  
enable  
info breakpoints  
tbreak
```

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

GDB basics - 5 Variables

Checking a variable

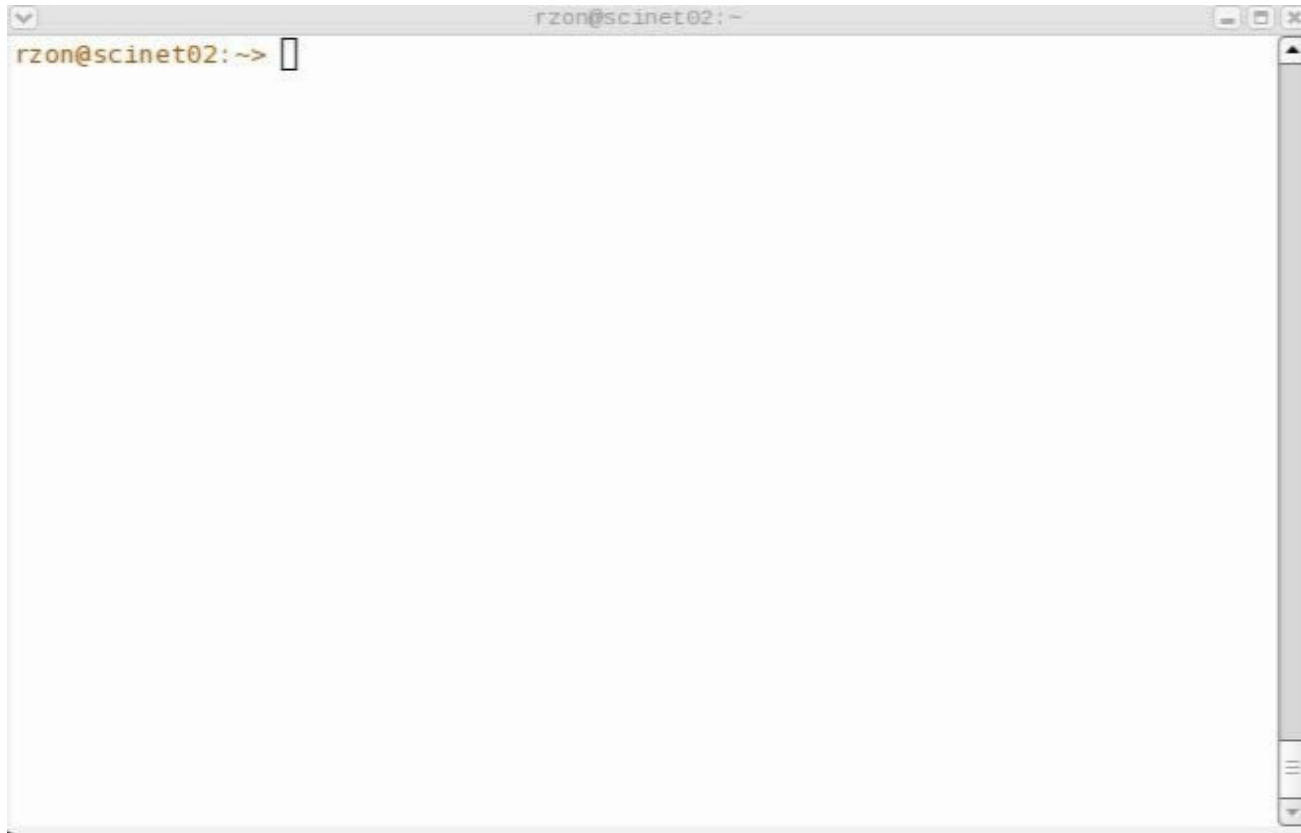
- Can print the value of a variable
- Can keep track of variable (print at prompt)
- Can stop the program when variable changes
- Can change a variable (“what if . . .”)

`gdb` commands

```
print  
display  
set variable  
watch
```

```
print variable  
print at every prompt  
change variable  
stop if variable changes
```

Demonstration gdb



A terminal window titled "rzon@scinet02: ~" with a shell prompt "rzon@scinet02: ~->". The window is empty except for the prompt and a cursor.

Graphical symbolic debuggers



Graphical symbolic debuggers

Features

- Nice, more intuitive graphical user interface
- Front to command-based tools: Same concepts
- Need graphics support. Requires tricks for compute nodes:

```
$ qsub .....  
$ checkjob <job-id>  
$ ssh -X -l <user> <your-node>
```

Available on SciNet

- **ddd**
\$ module load gcc ddd
\$ ddd <executable compiled with -g flag>
- **idb**
\$ module load intel java
\$ idb <executable compiled with -g flag>
- **ddt**
\$ module load ddt
(more later)

Graphical symbolic debuggers - ddd

DDD: /home/rzon/Courses/snugdebug/ex2/add.c (on gpc-f102n084)

File Edit View Program Commands Status Source Data Help

0: th

1: f 6.50046921 2: i 51 3: th 2

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

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

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

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

DDD: Threads (on gpc-f102n084)

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

Close Help

DDD

Run

Interrupt

Step Stepi

Next Nexti

Until Finish

Cont Kill

Up Down

Undo Redo

Edit Make

Graphical symbolic debuggers - idb

The screenshot displays the Intel(R) Debugger interface. The main window shows the source code for 'add.c' with the following content:

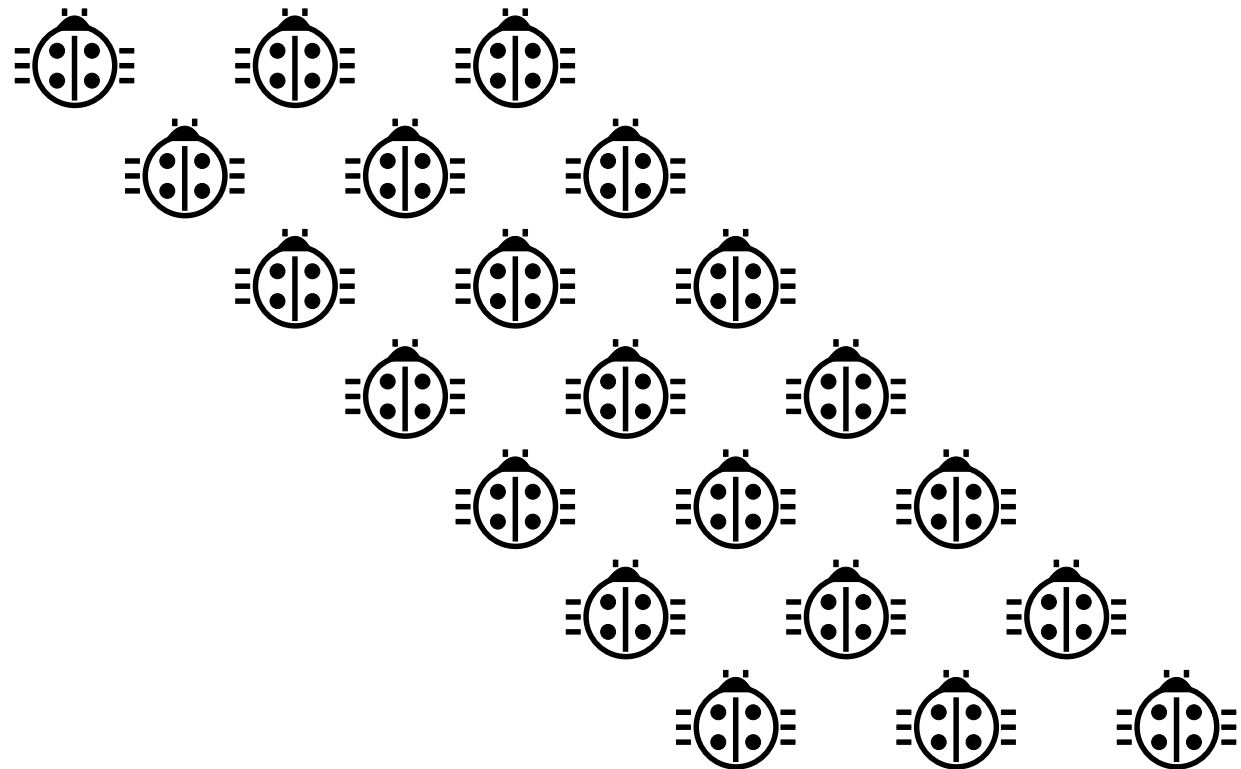
```
9   float f=0.0;
10  int i, th;
11 #pragma omp parallel for default(none) private(i,th) shared(f)
12  for (i = 0; i<100; i++) {
13      double g;
14      th = omp_get_thread_num();
15      printf("%d\n",th);
16      g = sqrt(0.25*i+th);
17      f += g;
18  }
19
20  printf("result = %f\n", f);
```

The 'Threads' window at the bottom shows the following table:

ID	Type	OS ID	Thread Library ID	Execution Attribute	Location
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)

The status bar at the bottom indicates the current thread: 0x000000000400887 in main.omp_fn.0 (omp_d..."/home/rzon/Courses/snugdebug/ex2/add.c":17

Parallel debugging



Parallel debugging

- Challenge: Simultaneous execution
- Shared memory:
 - OpenMP (Open Multi-Processing)
 - pthread (POSIX threads)
 - Private/shared variables
 - Intel compiler extra flag: `-debug parallel`
 - Race conditions
- Distributed memory:
 - MPI (Message Passing Interface)
 - Communication
 - Deadlock
- Hard to solve: some commercial debugger do a good job.
We've just obtained ddt licences!
But let's see how the command line ones handle it.

Parallel debugging - 1 Shared memory

gdb and idbc

- 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>`

idbc only

- Freezing/thawing thread
- Native OpenMP serialization (requires Intel compiler)
- Graphical: `ddd --debugger idbc`

```
info threads
thread
idb freeze/thaw t:[]
```

where is each thread?
change thread context
suspend thread(s)

Parallel debugging - Race conditions

helgrind

To find 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?
- 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.
3. Only then try full size.

Parallel debugging - 2 Distributed memory

padb

- Tool for debugging parallel mpi programs
- Requires openmpi and gdb:

```
module load gdb openmpi padb
```

Features

- Stack trace generation
- MPI Message queue display
- Deadlock detection and collective state reporting
- Process interrogation
- Signal forwarding/delivery
- MPI collective reporting
- Job monitoring

Parallel debugging - 2 Distributed memory

```
$ qsub -l nodes=1:ppn=8,walltime=1:00:00 -q debug -I
$ cd /scratch/where.ever
$ mpirun -np 16 whatever
$ padb --all --stack-trace --tree
Stack trace(s) for thread: 1
-----
[0-15] (16 processes)
-----
main() at ??:
  system_run() at ??:
    compute_forces() at ??:
      -----
      [8-15] (8 processes)
      -----
      IdVector_exchange() at ??:
        PMPI_Sendrecv() at ??:
          -----
          [8,10] (2 processes)
          -----
          ompi_request_default_wait() at ??:
            opal_progress() at ??:
              -----
              [9,11-15] (6 processes)
              -----
              mca_pml_ob1_send() at ??:
                opal_progress() at ??:
```

Parallel debugging - 2 Distributed memory

Advanced tricks

- You want `#proc` terminals with `gdb` for each process?
- Possible, but brace yourself!
- Small number of procs:
 - Start terminals: no x forwarding from compute nodes
 - Submit your job on scinet
 - Make sure its runs: `checkjob -v`
 - From each terminal, `ssh` into the appropriate nodes
 - Do `top` or `ps -C <exe>` to find process id (pid)
 - Attach debugger with `gdb -pid <pid>`.
 - This will interrupt the process (not for `idbc`).

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.

Note: You can use padb to find ranks of process etc.

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

Useful references

- G Wilson [Software Carpentry](http://software-carpentry.org/3_0/debugging.html)
http://software-carpentry.org/3_0/debugging.html
 - N Matloff and PJ Salzman
[The Art of Debugging with GDB, DDD and Eclipse](#)
 - [Padb](http://padb.pittman.org.uk): <http://padb.pittman.org.uk>
 - [Wiki](https://support.scinet.utoronto.ca/wiki): <https://support.scinet.utoronto.ca/wiki>
 - [Email](mailto:support@scinet.utoronto.ca): support@scinet.utoronto.ca
-