Debuggers and Parallel Debugging

HPC Best Practices

SciNet, Toronto
Debugging basics
Debugging basics

Help, my program doesn’t work!

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

↓

a miracle occurs

↓

My program works brilliantly!

$ icc -O3 answer.c
$ ./a.out
42

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

Debugging:

Methodical process of finding and fixing flaws in software
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...  \( \text{\textbf{There's a better way!}} \)
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 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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backtrace</td>
<td>function call stack</td>
</tr>
<tr>
<td>continue</td>
<td>continue</td>
</tr>
<tr>
<td>down</td>
<td>go to called function</td>
</tr>
<tr>
<td>up</td>
<td>go to caller</td>
</tr>
</tbody>
</table>
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 \texttt{--tui}

\begin{tabular}{|l|l|}
\hline
\textbf{gdb commands} & \\
\hline
\texttt{list} & list part of code \\
\texttt{next} & continue until next line \\
\texttt{step} & step into function \\
\texttt{finish} & continue until function end \\
\texttt{until} & continue until line/function \\
\hline
\end{tabular}
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 | unset breakpoint |
| condition | break if condition met |
| disable | disable breakpoint |
| enable | enable breakpoint |
| info breakpoints | list breakpoints |
| tbreak | 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

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>print variable</td>
</tr>
<tr>
<td>display</td>
<td>print at every prompt</td>
</tr>
<tr>
<td>set variable</td>
<td>change variable</td>
</tr>
<tr>
<td>watch</td>
<td>stop if variable changes</td>
</tr>
</tbody>
</table>
Demonstration gdb
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
Graphical symbolic debuggers - idb

```c
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);
```

```plaintext
ID | Type | OS ID | Thread Library ID | Execution Attribute | Location
---|------|-------|-------------------|---------------------|---------
$allthreads
1  | nati | 2606: 469387906204 | thawed | void main.omp.fn.0(void) */home/rzong/C
2  | nati | 2608: 1084623168 | thawed | void main.omp.fn.0(void) */home/rzong/C
3  | nati | 2608: 1113561408 | thawed | void main.omp.fn.0(void) */home/rzong/C
4  | nati | 2608: 1124551264 | thawed | <opaque> __write_nocancel(void)
```
Parallel debugging
Parallel debugging

- Challenge: Simultaneous execution
- Shared memory:
  - **OpenMP** (Open Multi-Processing)
  - **pthreads** (POSIX threads)
    - Private/shared variables
    - Intel compiler extra flag: `-debug parallel`
  - Race conditions
- Distributed memory:
  - **MPI** (Message Passing Interface)
    - Communication
    - Deadlock
- Hard to solve: some commercial 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: `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`

<table>
<thead>
<tr>
<th>info threads</th>
<th>where is each thread?</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>change thread context</td>
</tr>
<tr>
<td>idb freeze/thaw t:[]</td>
<td>suspend thread(s)</td>
</tr>
</tbody>
</table>
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:
  
  ```sh
  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\_obl\_send() at ?:\?
      opal\_progress() at ?:\?
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).
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

- N Matloff and PJ Salzman
  *The Art of Debugging with GDB, DDD and Eclipse*

- Padb: http://padb.pittman.org.uk

- Wiki: https://support.scinet.utoronto.ca/wiki

- Email: support@scinet.utoronto.ca