Debugging

Yaakoub El Khamra

Texas Advanced Computing Center
Debugging Outline

• GDB
  – Basic use
  – Attaching to a running job

• DDT
  – Identify MPI problems using Message Queues
  – Catch memory errors

• PTP
  – For the extremely patient*
Debugging

gdb and ddt
Why use a debugger?

• You’ve got code -> you’ve got bugs

• Buffered output (printf / write may not help)

• Fast & Accurate

• Many errors are difficult to find without one!
About GDB

GDB is the GNU Project DeBugger www.gnu.org/software/gdb/

From the GDB website: GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
Using GDB

Compile with debug flags: gcc -g -O0 ./srcFile.c

The \texttt{-g} flag generates the symbol table and provides the debugger with line-by-line information about the source code.

Execute debugger loading source dir: gdb -d srcDir ./exeFile

The \texttt{-d} option is useful when source and executable reside in different directories.

Use the \texttt{-q} option to skip the licensing message.

Type \texttt{help} at any time to see a list of the debugger options and commands.
Two levels of control

• Basic:
  – Run the code and wait for it to crash.
  – Identify line where it crashes.
  – With luck the problem is obvious.

• Advanced:
  – Set breakpoints
  – Analyze data at breakpoints
  – Watch specific variables
## GDB basic commands

<table>
<thead>
<tr>
<th>command</th>
<th>shorthand</th>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run/kill</td>
<td>r / k</td>
<td>NA</td>
<td>start/end program being debugged</td>
</tr>
<tr>
<td>continue</td>
<td>c</td>
<td>NA</td>
<td>continue running program from last breakpoint</td>
</tr>
<tr>
<td>step</td>
<td>s</td>
<td>NA</td>
<td>take a single step in the program from the last position</td>
</tr>
<tr>
<td>where</td>
<td>NA</td>
<td>NA</td>
<td>equivalent to backtrace</td>
</tr>
<tr>
<td>print</td>
<td>p</td>
<td>variableName</td>
<td>show value of a variable</td>
</tr>
<tr>
<td>list</td>
<td>l</td>
<td>srcFile.c:lineNumber</td>
<td>show the specified source code line</td>
</tr>
<tr>
<td>break</td>
<td>b</td>
<td>srcFile.c:lineNumber</td>
<td>set a breakpoint by line number or function name</td>
</tr>
<tr>
<td>watch</td>
<td>NA</td>
<td>variableName</td>
<td>stops when the variable changes value</td>
</tr>
</tbody>
</table>
GDB example

### divcrash.c

```c
#include <stdio.h>
#include <stdlib.h>
int myDiv(int, int);

int main(void)
{
    int res, x = 5, y;

    for(y = 1; y < 10; y++){
        res = myDiv(x,y);
        printf("%d,%d,%d\n",x,y,res);
    }
    return 0;
}

int myDiv(int x, int y){
    return 1/( x - y);
}
```

### divcrash.f90

```fortran
PROGRAM main

INTEGER :: myDiv
INTEGER :: res, x = 5, y

DO y = 1, 10
    res = myDiv(x,y)
    WRITE(*,*) x,y,res
END DO

END PROGRAM

FUNCTION myDiv(x,y)
    INTEGER, INTENT(IN) :: x, y
    myDiv = 1/(x-y)
    RETURN
END FUNCTION myDiv
```
GDB example

Compile the program and start the debugger:

% gcc -g -O0 ./divcrash.c
% gdb ./a.out

Start the program:

(gdb) run

The debugger will stop program execution with the following message:

Program received signal SIGFPE, Arithmetic exception.
0x0000000000040051e in myDiv (x=5, y=5) at divcrash.c:28
28 return 1/( x - y);

We can use gdb commands to obtain more information about the problem:

(gdb) where
#0 0x0000000000040051e in myDiv (x=5, y=5) at divcrash.c:28
#1 0x000000000004004cf in main () at divcrash.c:19
GDB example

In this case the problem is clear: a divide-by-zero exception happens in line 28 when variables \texttt{x} and \texttt{y} are the same.

This is related to the call to \texttt{myDiv} from line 19 that is within a for loop:

\begin{verbatim}
18: for(y = 1; y < 10; y++){
19:    res = myDiv(x,y);
\end{verbatim}

Eventually the loop sets the value of \texttt{y} equal to 5 (the value of \texttt{x}) producing the exception:

\begin{verbatim}
28: return 1/( x - y);
\end{verbatim}

With the problem identified we can kill the program and exit the debugger:
\begin{verbatim}
(gdb) kill
(gdb) quit
\end{verbatim}
Ex­am­in­ing data

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gdb) p x</td>
<td>(gdb) p x</td>
<td>Print scalar data x value</td>
</tr>
<tr>
<td>(gdb) p V</td>
<td>(gdb) p V</td>
<td>Print all vector V components</td>
</tr>
<tr>
<td>(gdb) p V[i]</td>
<td>(gdb) p V(i)</td>
<td>Print element i of vector V</td>
</tr>
<tr>
<td>(gdb) p V[i]@n</td>
<td>(gdb) p V(i)@n</td>
<td>Print n consecutive elements starting with V_i</td>
</tr>
<tr>
<td>(gdb) p M</td>
<td>(gdb) p M</td>
<td>Print all matrix M elements</td>
</tr>
<tr>
<td>(gdb) p M[i]</td>
<td>Not Available</td>
<td>Print row i of matrix M</td>
</tr>
<tr>
<td>(gdb) p M[i]@n</td>
<td>Not Available</td>
<td>Print n consecutive rows starting with row i</td>
</tr>
<tr>
<td>(gdb) p M[i][j]</td>
<td>(gdb) p M(i,j)</td>
<td>Print matrix element Mij</td>
</tr>
<tr>
<td>(gdb) p M[i][j]@n</td>
<td>(gdb) p M(i,j)@n</td>
<td>Print n consecutive elements starting with Mij</td>
</tr>
</tbody>
</table>

- No simple way to print columns in C or rows in Fortran
- Some debuggers print array slices (pgdbg, dbx), i.e. `p M(1:3,3:7)`
Breakpoint control

- Stop the execution of the program
- Allow you to examine the execution state in detail
- Can be assigned to a line or function
- Can be set conditionally

<table>
<thead>
<tr>
<th>command</th>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>breakpoints/b/br</td>
<td>Prints to screen all breakpoints</td>
</tr>
<tr>
<td>breakpoint</td>
<td>srcFile:lineNumber if a &lt; b</td>
<td>Conditional insertion of breakpoint</td>
</tr>
<tr>
<td>enable/disable</td>
<td>breakpointNumber</td>
<td>Enable/disable a breakpoint</td>
</tr>
<tr>
<td>delete</td>
<td>breakpointNumber</td>
<td>Delete a breakpoint</td>
</tr>
<tr>
<td>clear</td>
<td>srcFile:lineNumber functionName</td>
<td>Clear breakpoints at a given line or function</td>
</tr>
</tbody>
</table>
Attaching GDB to a running program

Use top to find out the PID of the tasks run by your program (in the top listing PIDs appear on the left, job names on the right).

% top

Attach \texttt{gdb} to the relevant PID:

% gdb \texttt{--p} \texttt{<PID>}

or:

% gdb
(gdb) attach \texttt{<PID>}

Once attached the debugger pauses execution of the program.

Same level of control than in a standard debugging session.
Attaching GDB to a running program

Best way to debug production runs. Don’t wait for your wall time to run out!

From the output of `qstat` obtain the node name where your code is running. In the `queue` field you will find an entry like

```
devdevelopment@i182-103.ta
```

- `queue name`
- `partial node name: i182-103.tacc.utexas.edu`
GDB Summary

• Compile using debug flags:
  % icc -g -O0 ./srcFile.c

• Run indicating the directory where the source is:
  % gdb -d srcDir ./exeFile

• Main commands:
  – run/kill
  – continue/next/step
  – break/watch
  – print
  – where
  – help
DDT: Parallel Debugger with GUI

Allinea Distributed Debugger Tool

- Multiplatform
- Supports all MPI distributions
- Capable of debugging large scale OMP/MPI
- Comprehensive
  - Memory checking
  - MPI message tracking
- Useful Graphical User Interface

www.allinea.com
DDT - Run

- General Options
- Queue Submission Parameters
- Processor and thread number
- Advanced Options
DDT - Run

- General Options
- Queue Submission Parameters
- Processor and thread number
- Advanced Options
- DDT 3.2
DDT – Queue Parameters

Each of these parameters may be changed

Project must be set!!
DDT: The debug session
DDT: Memory Leaks

Go to View -> Current Memory Usage

Process 0 is using much more memory than the others.

This looks like a memory leak.
DDT Summary

• ssh to Lonestar allowing X11 forwarding:
  % ssh -X username@<lonestar / stampede>.tacc.utexas.edu

• Compile with debugging flags:
  % mpicc -g -O0 ./srcFile.c

• Load the ddt module
  % module load ddt

• Run ddt
  % ddt ./exeFile

• Configure ddt properly before submission:
  – Options ➔ MPI version
  – Queue Parameters ➔ Wallclock/CPUs/Project
  – Advanced ➔ Memory Checking
Notes on Eclipse PTP

- Eclipse PTP is FREE
- Eclipse PTP is part of XSEDE: your tickets about Eclipse PTP will be answered by the Eclipse PTP developers (they are good about tickets)
- Eclipse PTP is a great tool to debug code locally (i.e. on your own workstation/laptop) before moving to production on XSEDE systems
- Eclipse PTP supports remote development, with existing configurations for all XSEDE machines including TACC systems
- Power IDE with refactoring, code completion, static analysis, collaborative code development…
```c
void
calc_pl(int rank, int num_procs)
{
    int 1;
    int num_intervals;
    double h;
    double mpi;
    double pi;
    double j;

    /* set number of intervals to calculate */
    if (rank == 0) {
        num_intervals = 100000000;
    }

    /* tell other tasks how many intervals */
    MPI_Bcast(&num_intervals, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* now everyone does their calculation */
    h = 1.0 / (double) num_intervals;
    sum = 0.0;
    for (i = rank + 1; i <= num_procs; i += num_procs) {
        ...
MPI_PLC (C/C++ Application)

/* find out number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

if (my_rank == 0) {
    // some code
} else {
    // some code
}

Console:
MPI_PLC (C/C++ Application)
[139576,1,0]: A high-performance open MPI point-to-point messaging module
was unable to find any relevant network interfaces:
Module: OpenFabrics (openib)
    Host: dauntless.tacc.utexas.edu

Another transport will be used instead, although this may result in
lower performance.
Choose Resource Manager Type
Select the type of resource manager to use

### Resource Manager Types:
- Grid Engine-Generic-Batch
- IBM LoadLeveler
- IBM LoadLeveler (Blue Gene)
- IBM Parallel Environment
- IBM Parallel Environment (proxy)
- MPICH2
- MPICH2-Generic-Interactive
- Open MPI
- Open MPI-Generic-Interactive
- PBS-BGP-Batch
- PBS-BGQ-Batch
- PBS-Generic-Batch
- PBS-Generic-Interactive
- Remote Launch
- Remote-Generic-Interactive
- SLURM
- SLURM-BGP-Batch
- SLURM-BGQ-Batch
- SLURM-Generic-Batch
- Torque-Generic-Batch
- Torque-Generic-Interactive
- edu.illinois.ncsa.bluewaters:pbs.batches.xe
- edu.illinois.ncsa.bluewaters:pbs.batches.xk
- edu.illinois.ncsa.forsyte:pbs.batches.mpich2
- edu.illinois.ncsa.forsyte:pbs.batches.openmpi
- edu.illinois.ncsa.forsyte:pbs.interactive.openmpi
- edu.sdsc.trestles:pbs.batches.xe
- edu.texas.tacc.ironman:pbs.batches.xe
- edu.utk.nics.keeneland:pbs.batches
- edu.utk.nics.kraken:pbs.batches
- govani.alcf:bgp:pbs.batches
- govani.alcf:bgq:pbs.batches
Debugging: Conclusions

• You ALWAYS want to debug your code if:
  o Program exhibits erratic, random behavior when you are not using random variables in control statements (hint: memory corruption/stomping)
  o Program appears to "hang" somewhere: attach a debugger and see "where" it is stuck. Profiling will NOT help with "stuck" program flow
  o Program exits way too fast (and you do not have exit error messages built into your code, which you want to consider)
  o Wrong/bad results: unless you are working with infinite or semi-infinite Cantor sets, or attempting to use GiNaC, you should not be getting NaN's or Inf's in your variables: add conditional watch expressions to halt execution when you encounter these values (or use isnan, isinf functions in C)

• This is not to say that you cannot debug your code with print statements. Whatever gets your code operating at designed parameters: performance, readability, maintainability, rapid development, informative crashes/bug-free/idiot-proof etc... that is what you want to be doing. Debuggers get you there quicker