Introduction to debugging

Martin Čuma
Center for High Performance Computing University of Utah
m.cuma@utah.edu

Overview

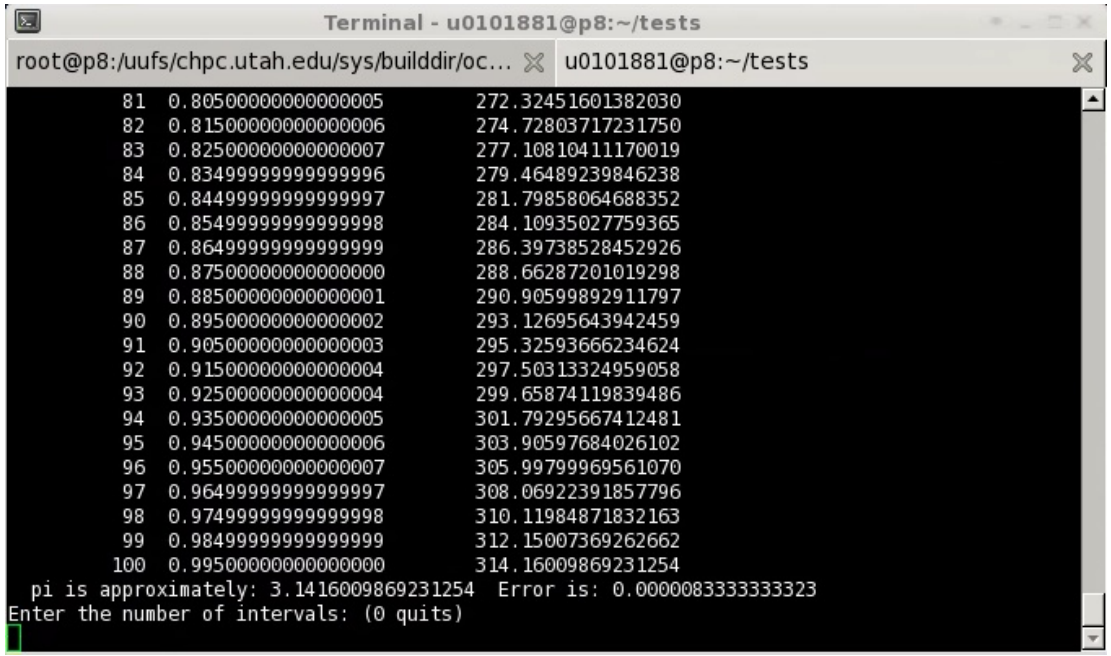
- Program errors
- Simple debugging
- Graphical debugging
- DDT and Totalview
- Intel tools
- Interpreted languages

Program errors

- crashes
  - segmentation faults (bad memory access)
    - often writes core file – snapshot of memory at the time of the crash
  - wrong I/O (missing files)
  - hardware failures
- incorrect results
  - reasonable but incorrect results
  - NaNs – not a numbers – division by 0, …

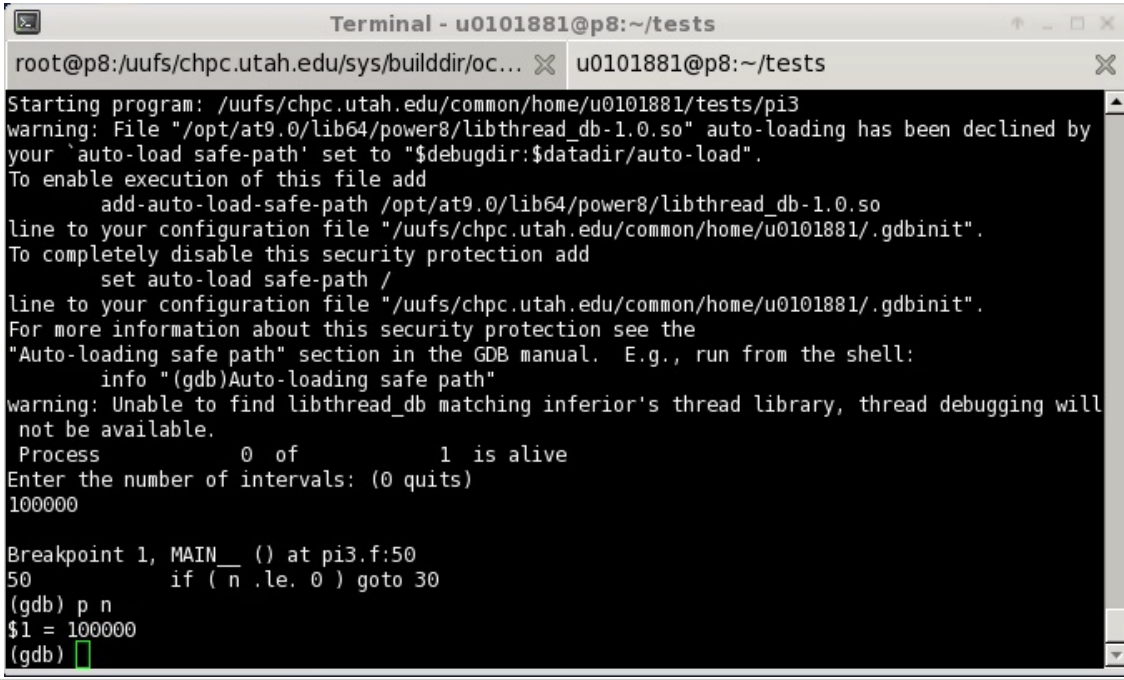
write/printf

- write variables of interest into the stdout or file
- simplest but cumbersome
  - need to recompile and rerun
  - need to browse through potentially large output



Terminal debuggers

- text only, e.g. gdb, idb
- need to remember commands or their abbreviations
- need to know lines in the code (or have it opened in other window)
- useful for quick code checking on compute nodes and core dump analysis



GUI debuggers

- have graphical user interface
- freeware or commercial
- Eclipse CDT - free
- PGI's pdbg – part of PGI compiler suite
- Intel development tools
- Rogue Wave Totalview - commercial
- Allinea DDT - commercial

Totalview and DDT

- The only real alternative for parallel or accelerator debugging
- Cost a lot of money (thousands of $), but, worth it
- We had Totalview license (for historical reasons), 32 tokens enough for our needs (renewal ~$1500/yr)
- In 2017 we switched to DDT which gave us competitive upgrade
- XSEDE systems have DDT

How to use DDT

1. Compile binary with debugging information
   - flag -g

```
gcc -g test.f -o test
```

2. Load module and run DDT

```
module load ddt
```

   - DDT + executable

```
ddt ./executable
```

   - DDT + core file

```
ddt executable core_file
```

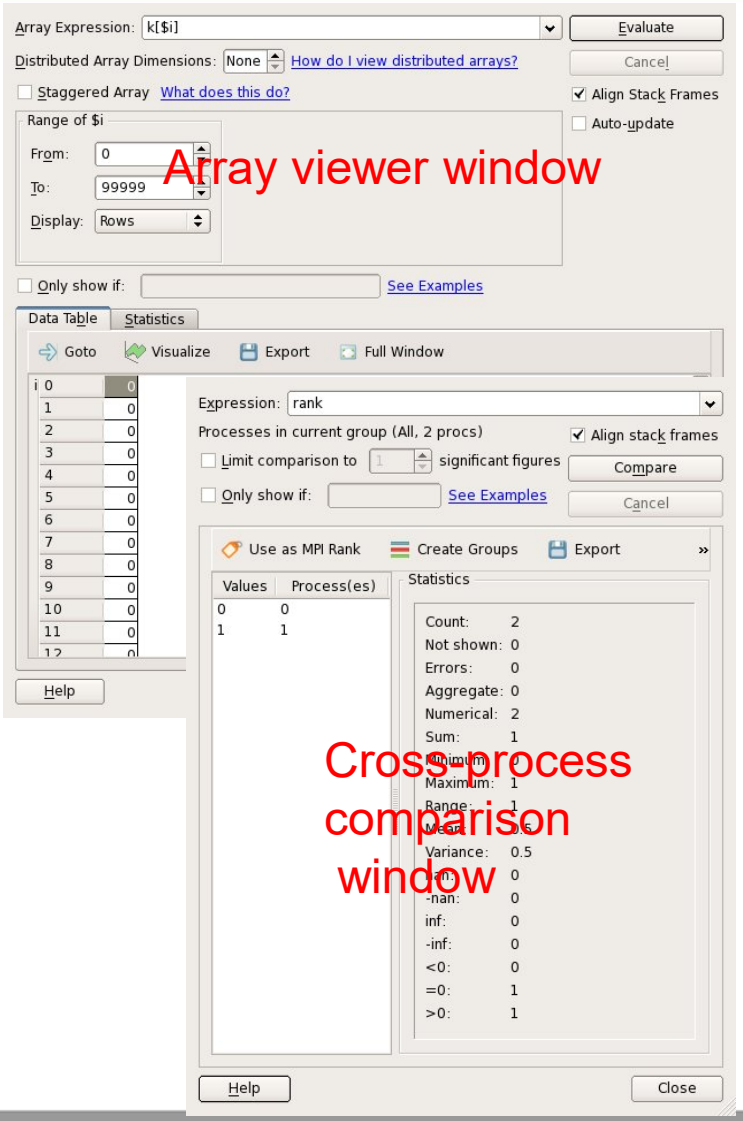
How to use DDT

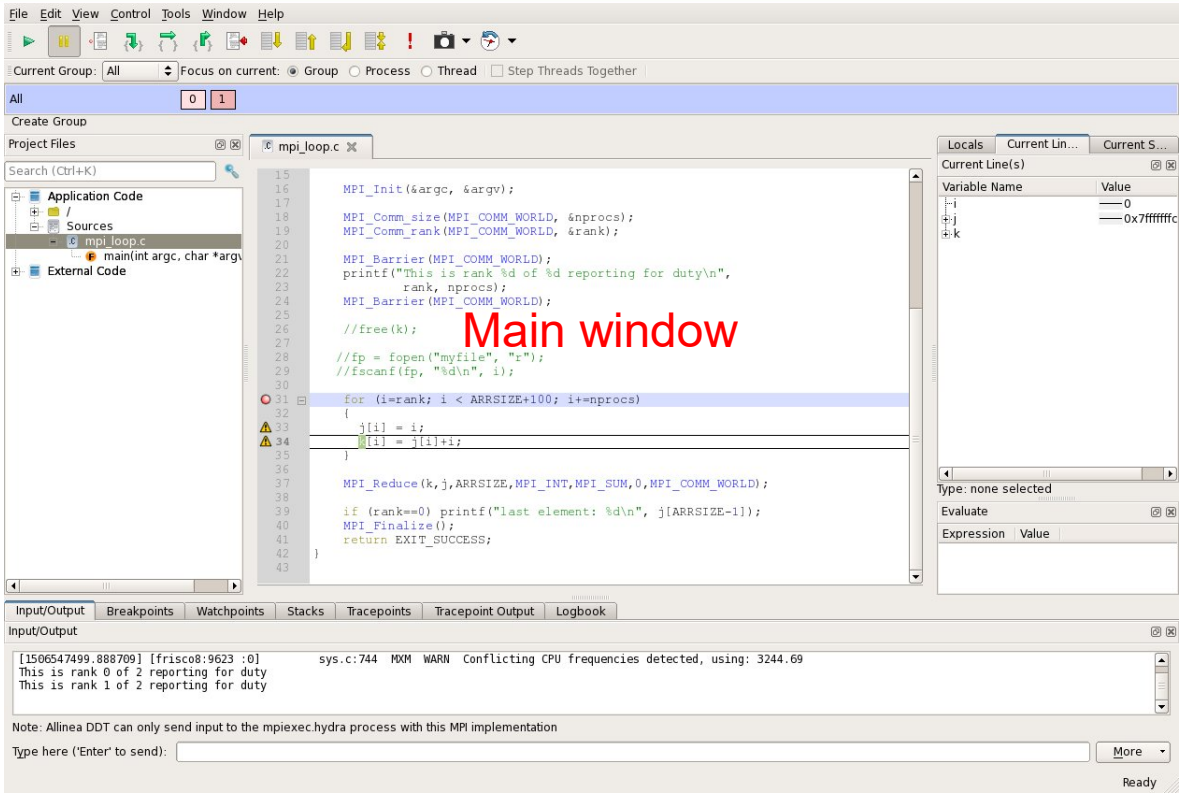
- run DDT and attach the executable
  - start DDT
  - fill in information in the Run dialog
  - click Run button to start the program
- run DDT and attach running program
  - start DDT
  - click “Attach to an already running program”
  - choose process and click “Attach to” button

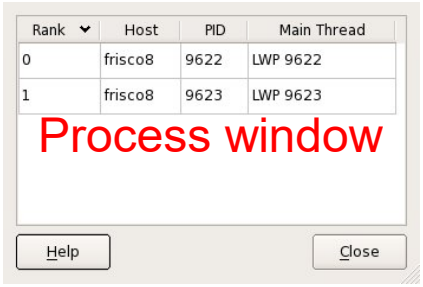
3. DDT operation

- left mouse button - select
- right mouse button – context sensitive menu

DDT windows


Array Expression: k[$i]
Evaluate
Distributed Array Dimensions: None
How do I view distributed arrays?
Cancel
Staggered Array
What does this do?
Align Stack Frames
Auto-update
Range of $i
From: 0
To: 99999
Display: Rows
Array viewer window
Only show if:
See Examples
Data Table
Statistics
Goto
Visualize
Export
Full Window
Expression: rank
Processes in current group (All, 2 procs)
Align stack frames
Limit comparison to
significant figures
Compare
Only show if:
See Examples
Cancel
Use as MPI Rank
Create Groups
Export
Values
Process(es)
Statistics
Count: 2
Not shown: 0
Errors: 0
Aggregate: 0
Numerical: 2
Sum: 1
Minimum: 0
Maximum: 1
Range: 1
Mean: 0.5
Variance: 0.5
nan: 0
-nan: 0
inf: 0
-inf: 0
<0: 0
=0: 1
>0: 1
Help
Close
Cross-process comparison window


File Edit View Control Tools Window Help
Current Group: All
Focus on current: Group Process Thread
Step Threads Together
Create Group
Project Files
mpi_loop.c
Application Code
Sources
External Code
Locals
Current Lin...
Current S...
Variable Name
Value
Main window
Input/Output
Breakpoints
Watchpoints
Stacks
Tracepoints
Tracepoint Output
Logbook
Type here ('Enter' to send):
More
Ready


Rank
Host
PID
Main Thread
0
frisco8
9622
LWP 9622
1
frisco8
9623
LWP 9623
Process window
Help
Close

DDT basic operations

- Data examination
  - view data in the variable windows
  - change the values of variables
  - modify display of the variables
  - visualize data
- Action points
  - breakpoints and barriers (static or conditional)
  - watchpoints and tracepoints
  - evaluation of expressions

Multiprocess debugging

- Automatic attachment of child processes
- Create process groups
- Share breakpoints among processes
- Process barrier breakpoints
- Process group single-stepping
- View variables across procs/threads
- Display MPI message queue state

Basic operation example

- Load up an existing program
  - DDT windows
  - step through the code
  - place breakpoints
  - examine variables
- Load a core file
  - examine the crash point

Process view window

- Stack trace – procedure hierarchy - left
- Stack frame – variables display - right
- Source code – code + process navigation - center
- Threads list – in case of multithreaded application - top
- Action points – list of breakpoints, barriers,… - bottom

Action points

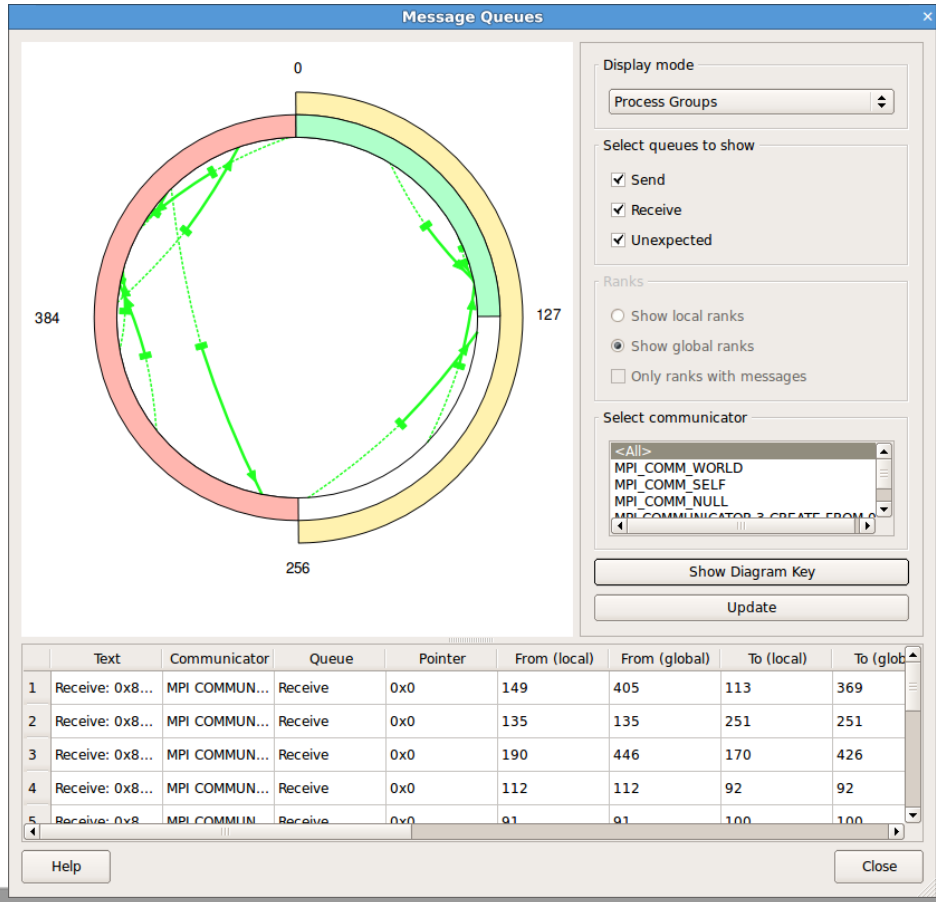
- Breakpoints and barriers
  - toggle location with left mouse (shift for barrier)
  - right-click – Properties for options
  - option conditional breakpoints
- Watchpoints
  - watch for change in a memory location
- Tracepoints
  - write screen output whenever line is executed

Data examination

- Scalars in variable view on the right of the main window
- Arrays in Array viewer
  - right click on a variable, choose Array viewer
  - adjust array dimensions, hit evaluate
  - filter array values, e.g. $value > 0
- variable visualization, statistics,…

Parallel debugging

- Process/thread groups
- Cross-process/thread data comparison
- Message queue state graph and display



Memory debugging

- Enable in the initial Run dialog
  - Memory usage
  - Memory leak detection
  - Exceeding array bounds
  - Dangling pointers

Accelerator debugging

- We license the GPU add-on to DDT
- Works well with CUDA
- With OpenACC, the debugger sees what the compiler generates so stepping through code may not correspond to actual source lines
  - Use breakpoints and examine data at the breakpoints

Python debugging

- Can debug Python modules that are written in C/C++ or Fortran
- Or mixed C/C++, Fortran and Python code
- No stepping, breakpoints, variable view of the Python code

Some useful resources

- DDT webpage

https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt

- Setting up DDT

  Clusters: `module load ddt`

  Some group desktops: `inquire at CHPC`

- Documentation

https://developer.arm.com/docs/101136/latest/ddt

https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/resources/videos

Code checkers

- compilers check for syntax errors
  - some compiler flags help too (-C)
- memory checking tools - many errors are due to bad memory management
  - valgrind – easy to use
  - purify – harder to use

Intel software development products

- We have a 2 concurrent user license
- Tools for all stages of development
  - Compilers and libraries
  - Verification tools
  - Profilers
- More info

https://software.intel.com/en-us/intel-parallel-studio-xe

Intel Inspector

- Thread checking
  - Data races and deadlocks
- Memory checker
  - Like leaks or corruption
  - Good alternative to Totalview MemoryScape or DDT
- Standalone or GUI integration
- More info

http://software.intel.com/en-us/inspector/

Intel Inspector

- Source the environment

  `module load inspectorxe`

- Compile with `-tcheck -g`

  `ifort -openmp -tcheck -g trap.f`

- Run tcheck

  `inspxe-gui` – graphical user interface

  `inspxe-cl` – command line

- Tutorial

https://software.intel.com/en-us/articles/inspector-tutorials

Intel Trace Analyzer and Collector

- MPI profiler and correctness checker
- Detects violations of MPI standard and errors in execution environment
- To use correctness checker

```
module load intel impi itac
setenv VT_CHECK_TRACING 0
mpirun -check-mpi -n 4 ./myApp
```

- ITAC documentation

https://software.intel.com/en-us/trace-analyzer/training

Interpreted languages debugging

- Matlab
  - Debugger built in the editor
- Python
  - gdb like debugging built in (pdb module)
  - Many different third party IDEs
  - PyCharm, but beware of larger memory/CPU use if debugging big data
- R
  - gdb like debugging built in (traceback(), browse(), debug())
  - Rstudio allows debugging

Conclusions

- Terminal debuggers
- Compiler vendor debuggers
- DDT/Totalview for graphical debugging
- Code checkers and memory checkers
- Inspector for thread and memory debugging
- ITAC MPI checker
- Interpreted languages debugging