Parallel Debugging Tools

New User Training 2017

Woo-Sun Yang
User Engagement Group, NERSC

February 23, 2017
Debugging

• Why debugging?
  – Your program crashes for a unknown reason
  – Your program gives wrong results

• How to find coding errors?
  – Using print statements
    • Insert print statements in strategic locations
    • Can be difficult to know where the code fails and whether variables have incorrect values
    • Recompile whenever you make a change - tedious and time-consuming
  – Using debuggers
    • You compile only once (generally)
    • Can point to where the code fails
    • They let you control execution pace of your program and examine variables
    • Useful tools can aid your detective work greatly
      – Visualization and statistics
      – Memory debugging
      – MPI message queue
Parallel debuggers on Cori and Edison

- **Parallel debuggers with a graphical user interface**
  - DDT (Distributed Debugging Tool)
  - TotalView

- **Specialized debuggers on Cori and Edison**
  - STAT (Stack Trace Analysis Tool)
    - Collect stack backtraces from all (MPI) tasks
  - ATP (Abnormal Termination Processing)
    - Collect stack backtraces from all (MPI) tasks when an application fails

- **Valgrind**
  - Suite of debugging and profiling tools
DDT and TotalView

• **GUI-based traditional parallel debuggers**
  – Intuitive and simple to use; many useful tools
  – Allow to control program’s execution pace and, sometimes, execution path
    • Set breakpoints, watchpoints and tracepoints
  – Display the values of variables and expressions, and visualize arrays
    • Check whether the program is executing as expected
  – Memory debugging
  – Message queue feature **NOT** working with Cray MPI

• **Works for C, C++, Fortran programs with MPI, OpenMP, pthreads**
  – DDT supports CAF (Coarray Fortran) and UPC (Unified Parallel C), too

• **Maximum application size for the debuggers at NERSC**
  – DDT: up to 4096 MPI tasks on Cori (Haswell and KNL) and Edison
  – TotalView: up to 512 MPI tasks on Cori (Haswell) and Edison
  – Licenses shared among users and machines

• **For info**
How to build and run with DDT

$ ftn -g -O0 -o jacobi_mpi jacobi_mpi.f90  
Compile with -g to have debugging symbols
Include -O0 for the Intel compiler

$ salloc -N 1 -t 30:00 -p debug -C knl,quad,cache  
Start an interactive batch session

$ module load allineatools  
Load the allineatools module to use DDT

$ ddt ./jacobi_mpi  
Start DDT

The module name will change to ‘forge’ for future versions
If you are far away from NERSC

- Remote X window application (GUI) over network: slow response

- Two solutions
  - Use NX to improve the speed
    - Works with any X window applications
  
  - Use Allinea Forge remote client
    - Runs on your desktop/laptop
    - Submit a debugging batch job from a NERSC machine and make the client **reverse connect** to the job
    - Displays results in real time
    - No license file required on your local desktop/laptop
Using NX
(1) Select ‘Configure’ to create a configuration for a NERSC machine

2nd entry for a MOM node
Cori: cmom02 or cmom06
Edison: edimom01, ..., or edimom06

(2) Create a configuration

Note that the paths will change for future versions
(3) Select a machine

RUN
Run and debug a program.

ATTACH
Attach to an already running program.

OPEN CORE
Open a core file from a previous run.

MANUAL LAUNCH (ADVANCED)
Manually launch the backend yourself.

OPTIONS

Remote Launch:
- Off
- Configure...
  - cori
  - edison
  - carl

(4) Enter the NIM password

Connect to remote host

Password: [enter password]
(5) Submit a batch job on a NERSC machine and start DDT

$ s alloc -N 1 -t 30:00 -p debug -C knl ...
$ module load allineatools
$ d dt --connect ./jacobi_mpiomp

(6) Accept the request

A new Reverse Connect request is available from nid08791 for Allinea DDT.
Command Line: --connect ./jacobi_mpiomp
Do you want to accept this request?

(7) Set parameters and run

![Image of a DDT interface with parameters set and a run button]
For navigation

Parallel stack frame view is helpful in quickly finding out where each process is executing.

Processing entity to control

To check the value of a variable, right-click on a variable or check the pane on the right.

Sparklines to quickly show variation over MPI tasks
• **Play/Continue**
• **Pause**
• **Add Breakpoint**
• **Step Into**
  – To next line; if it’s a function call, enter the function
• **Step Over**
  – To next line in the current stack frame even if it’s a function call
• **Step Out**
  – Return to the caller function
• **Run To Line**
Breakpoints, watchpoints and tracepoints

• **Breakpoint**
  – Stops execution when a selected line (breakpoint) is reached
  – Double click on a line to create one; there are other ways, too

• **Watchpoints for variables or expressions**
  – Stops when a variable or an expression changes its value

• **Tracepoints**
  – When reached, prints what lines of codes is being executed and the listed variables

• **Can add a condition for an action point**
  – Useful inside a loop

• **Can be active or inactive**
Many ways to check variables

- Right click on a variable for a quick summary
- Variable pane
- Evaluate pane
- Display variable values over processes (Compare across processes) or threads (Compare across threads)
- MDA (Multi-dimensional Array) Viewer
  - Visualization
  - Statistics
Memory debugging

• **Why?**
  – To detect memory leaks
  – To catch out-of-bound array references
  – To catch other memory errors (“double free”, etc.)
  – To see memory usage

• **For a statically-linked executable**
  – For non-threaded code
    
    $ ftn -c -g -O0 myprog.f$
    $ static_linking_ddt_md ftn -o myprog myprog.o$
    
    # instead of $ ftn -o myprog myprog.o$
  – `static_linking_ddt_md_th` for threaded program
  – Similarly for C and C++ codes
    
    – `static_linking_ddt_md` and `static_linking_ddt_md_th` are utility scripts provided by NERSC

• **For a dynamically-linked executable, build as usual**
Enabling memory debugging

- **For a dynamically-linked binary only**
  - Check ‘Preload the memory debugging library’
  - Select the appropriate one from the ‘Language’ pull-down menu

- **Adding guard pages (default: 4 KB) before or after memory blocks for detecting out-of-bound heap array references**

When you click ‘Details…’
Memory debugging – Overall Memory Stats

Tools > Overall Memory Stats

memory_leaks.f from NERSC DDT web page
KNL MCDRAM usage on Cori

- Memory blocks allocated in MCDRAM with memkind’s `hbw_malloc` calls and Fortran’s `fastmem` directives are annotated accordingly in DDT/7.0.
With `numactl`

- In an interactive batch job:
  1. Run `ddt` in background
     
     
     
     
     
     \$ ddt &
  2. Select ‘MANUAL LAUNCH (ADVANCED)’
  3. Set run parameters and check ‘Memory Debugging’
  4. Click ‘Listen’
  5. Run a `srun` command:
     
     
     
     
     \$
     \$ srun --n ... numactl \ 
     \--preferred=1 \ 
     \ allinea-client ./a.out

- `--mem_bind=...`: simply use `srun`’s
  \- `--mem_bind=map_mem:...` instead

- MCDRAM usage is not properly annotated in version 7.0. Reported to Allinea.
TotalView

$ salloc -N 1 -t 30:00 -p debug
$ module load totalview
$ export OMP_NUM_THREADS=6
$ totalview srun -a -n 4 ./jacobi_mpiomp

Then,

• Click OK in the ‘Startup Parameters - srun’ window
• Click ‘Go’ button in the main window

• Click ‘Yes’ to the question ‘Process srun is a parallel job. Do you want to stop the job now?’
To see the value of a variable, right-click on a variable to “dive” on it or just hover mouse over it.

State of MPI tasks and threads; members denoted roughly as ‘rank.thread’

Breakpoints, etc.

For selecting MPI task and thread

For navigation
Viewing variables

• Variable window

• Visualization and stats

Tools > Visualize

Tools > Statistics
Memory debugging with MemoryScape

• MemoryScape integrated into TotalView for memory debugging
  – Memory leaks
  – Memory usage
  – Memory corruption
  – ...

• A statically-linked executable

  $ module load totalview
  $ CC -g -O0 -o memory_leaks memory_leaks.o ${TVMEMDEBUG_POST_OPTS}

• A dynamically-linked executable, build as usual

  $ CC -dynamic -g -O0 -o memory_leaks memory_leaks.o
Memory debugging with MemoryScape

• Start TotalView and enable memory debugging in the ‘Startup Parameters’ window
• Proceed to use TotalView as usual

• For memory-related issues, open MemoryScape from the Debug pull-down menu
Memory debugging examples

Corrupted guard blocks
STAT (Stack Trace Analysis Tool)

• Gathers stack backtraces (showing the function calling sequences leading up to the ones in the current stack frames) from all (MPI) processes and merges them into a single file (*.dot)
  – Results displayed graphically as a call tree showing the location in the code that each process is executing and how it got there
  – Can be useful for debugging a hung application
  – With the info learned from STAT, can investigate further with DDT or TotalView

• Works for MPI, CAF and UPC, but not OpenMP

• STAT commands (after loading the ‘stat’ module)
  – stat-cl: invokes STAT to gather stack backtraces
  – stat-view: a GUI to view the results
  – stat-gui: a GUI to run STAT or view results

• For more info:
Hung application with STAT

- If your code hangs in a consistent manner, you can use STAT to see if and where some MPI ranks are stuck.
- Currently, one known way to use STAT is as follows.

  $ ftn -g -o jacobi_mpi jacobi_mpi.f90
  $ salloc -N 1 -t 30:00 -p debug -C knl,quad,cache
  ...
  $ srun -n 4 ./jacobi_mpi &
  [1] 93834
  $ module load stat
  $ stat-cl -i 93834

- Attaching to application...
  Attached!
- Application already paused... ignoring request to pause
- Sampling traces...
  Traces sampled!
  ...
- Resuming the application...
  Resumed!
- Merging traces...
  Traces merged!
- Detaching from application...
  Detached!

Results written to /global/cscratch1/sd/wyang/debugging/stat_results/jacobi_mpi.0001

  $ ls -l stat_results/jacobi_mpi.0001/*
  -rw-r----- 1 wyang wyang 2768 Feb 20 21:24 stat_results/jacobi_mpi.0001/00_jacobi_mpi.0001.3D.dot
  $ stat-view stat_results/jacobi_mpi.0001/00_jacobi_mpi.0001.3D.dot
Hung application with STAT (Cont’d)

Ranks 1 & 2 are here
Rank 0 is here
Rank 3 is here
ATP (Abnormal Termination Processing)

• ATP gathers stack backtraces from all processes if an application fails
  – Invokes STAT underneath
  – Output in atpMergedBT.dot and atpMergedBT_line.dot (which shows source code line numbers), which are to be viewed with stat-view

• By default, the atp module is loaded on Cori and Edison, but ATP is not enabled; to enable:
  
  ```sh
  export ATP_ENABLED=1  # sh/bash/ksh
  setenv ATP_ENABLED 1    # csh/tcsh
  ```

• Can get core dumps (core.atp.jobid.rank), too, by setting coredumpsize unlimited:
  
  ```sh
  ulimit -c unlimited    # sh/bash/ksh
  unlimit coredumpsize  # csh/tcsh
  ```

  but they do not represent the exact same moment in time (therefore the location of a failure can be inaccurate)

• For more info
  – ‘intro_atp’ man page
Hung application with ATP

- Force to generate backtraces from a hung application
- For the following to work, must have used
  - ‘export ATP_ENABLED=1’ in batch script
  - ‘export FOR_IGNORE_EXCEPTIONS=true’ in batch script for Intel Fortran
  - ‘-f no-backtrace’ at compile/link time for GNU Fortran

$ sacct -j 4097861 Find the job step ID

<table>
<thead>
<tr>
<th>JobID</th>
<th>JobName</th>
<th>Partition</th>
<th>Account</th>
<th>AllocCPUS</th>
<th>State</th>
<th>ExitCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$ ssh edimom02 Kill the application on a MOM node
$ scancel -s ABRT 4097861.0
$ exit
$ cat slurm-4097861.out

Application 4097861 is crashing. ATP analysis proceeding...

... Process died with signal 6: 'Aborted'

View application merged backtrace tree with: stat-view atpMergedBT.dot

... $ module load stat
$ stat-view atpMergedBT.dot  # or statview atpMergedBT_line.dot
Valgrind

• Suite of debugging and profiler tools
• Tools include
  – **memcheck**: memory error and memory leaks detection
  – **massif, dhat (exp-dhat)**: heap profilers
  – **cachegrind**: a cache and branch-prediction profiler
  – **callgrind**: a call-graph generating cache and branch prediction profiler
  – **helgrind, drd**: pthreads error detectors

• For info:
Valgrind’s memcheck

$ module load valgrind
$ ftn -dynamic -g -O0 memory_leaks.f $VALGRIND_MPI_LINK
$ salloc -N 1 -t 30:00 -p debug -C knl
$ srun -n 2 valgrind --leak-check=full --log-file=%p ./a.out
$ ls -l
...
-rw-r--r--  1 wyang   wyang  7550 Feb 21 23:36 91835
-rw-r--r--  1 wyang   wyang  7550 Feb 21 23:36 91836

• Let’s look at the report for process 91835

$ more 91835
...
==91835== LEAK SUMMARY:
==91835==     definitely lost: 83,886,880 bytes in 20 blocks
==91835==     indirectly lost: 0 bytes in 0 blocks
==91835==      possibly lost: 41,943,440 bytes in 10 blocks
==91835==    still reachable: 103,903 bytes in 74 blocks
==91835==         suppressed: 0 bytes in 0 blocks
...

• Can suppress spurious error messages by using a suppression file
  (--suppressions=/path/to/directory/file)

Could have explicitly added ‘--tool=memcheck’
Valgrind's massif

- For profiling heap memory usage

```bash
$ ftn -g -O2 memory_leaks.f
$ srun -n 2 -c 128 valgrind --tool=massif ./a.out
$ ls -lrt
...
-rw------- 1 wyang wyang 50233 Feb 21 23:55 massif.out.92841
-rw------- 1 wyang wyang 81113 Feb 21 23:55 massif.out.92842
$ ms_print massif.out.92841
...
```

For profiling heap memory usage:

- `@`: detailed snapshot where detailed info is provided
- `#`: peak snapshot where the peak heap usage is

This example strongly suggests memory leaks.
Valgrind's massif (Cont'd)

<table>
<thead>
<tr>
<th>n</th>
<th>time(i)</th>
<th>total(B)</th>
<th>useful-heap(B)</th>
<th>extra-heap(B)</th>
<th>stacks(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>531,809,757</td>
<td>96,862,856</td>
<td>96,761,707</td>
<td>101,149</td>
<td>0</td>
</tr>
<tr>
<td>91</td>
<td>658,233,924</td>
<td>126,259,976</td>
<td>126,130,750</td>
<td>129,226</td>
<td>0</td>
</tr>
</tbody>
</table>

99.90% (126,130,750B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.

-->99.66% (125,830,320B) 0x4E3FF6A: _mm_malloc (in /opt/intel/compilers_and_libraries_2017.1.132/linux/compiler/lib/intel64_lin/libintlc.so.5)

| | -->99.66% (125,830,320B) 0x40AF1F: for_allocate (in /global/cscratch1/sd/wyang/debugging/memory_leaks) |
| | | -->33.22% (41,943,440B) 0x4033AF: MAIN__ (memory_leaks.f:41) |
| | | | -->33.22% (41,943,440B) 0x402FDC: main (in /global/cscratch1/sd/wyang/debugging/memory_leaks) |
| | | | -->33.22% (41,943,440B) 0x403621: MAIN__ (memory_leaks.f:51) |
| | | | | -->33.22% (41,943,440B) 0x402FDC: main (in /global/cscratch1/sd/wyang/debugging/memory_leaks) |
| | | | | -->33.22% (41,943,440B) 0x403898: MAIN__ (memory_leaks.f:54) |
| | | | | | -->33.22% (41,943,440B) 0x402FDC: main (in /global/cscratch1/sd/wyang/debugging/memory_leaks) |
| | | -->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%) |
| | | -->00.24% (300,430B) in 1+ places, all below ms_print's threshold (01.00%) |

<table>
<thead>
<tr>
<th>n</th>
<th>time(i)</th>
<th>total(B)</th>
<th>useful-heap(B)</th>
<th>extra-heap(B)</th>
<th>stacks(B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>94</td>
<td>658,456,870</td>
<td>126,056,640</td>
<td>125,935,407</td>
<td>121,233</td>
<td>0</td>
</tr>
</tbody>
</table>
National Energy Research Scientific Computing Center