Debugging Tools

Woo-Sun Yang
NERSC User Services Group

NUG Training
February 3, 2014
Debuggers on NERSC machines

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

• **Specialized debuggers on Hopper 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
  - CCDB (Cray Comparative Debugger)
    - Comparative debugging

• **Valgrind**
  - Suite of debugging and profiler tools
DDT and TotalView

• GUI-based traditional parallel debuggers
  – Control program’s execution pace by it advance to a desired location
  – Set breakpoints, watchpoints and tracepoints
  – Display the values of variables and expressions, and visualize arrays
    • To check whether the program is executing as expected
  – Memory debugging
  – And more...

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

• Maximum application size to use the debuggers at NERSC
  – DDT: up to 8192 MPI tasks
  – TotalView: up to 512 MPI tasks
  – Licenses shared among users

• For info
  – $ALLINEA_TOOLS_DOCDIR/userguide.pdf (after loading the ‘allineatools’ module)
  – http://www.roguewave.com/products/totalview
% ftn -g -O0 -o jacobi_mpi jacobi_mpi.f90  Compile with -g to have debugging symbols
% qsub -IV -lmppwidth=24,walltime=30:00 -q debug  Include -O0 for the Intel compiler
% cd $PBS_O_WORKDIR  Start an interactive batch session
% module load allineatools  Load the allineatools module to use DDT
% ddt ./jacobi_mpi  Start DDT
DDT (cont’d)

Navigate using these buttons

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

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
% qsub -IV -lmpwidth=24,walltime=30:00 -q debug
% cd $PBS_O_WORKDIR
% module load totalview
% totalview aprun -a -n 4 ./jacobi_mpi

Then,
- Click OK in the ‘Startup Parameters - aprun’ window
- Click ‘Go’ button in the main window
- Click ‘Yes’ to the question ‘Process aprun 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.
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 (STAT or stat-cl): invokes STAT to gather stack backtraces
  - statview (STATview or stat-view): a GUI to view the results
  - statgui (STATGUI or stat-gui): a GUI to run STAT or view results

- For more info:
  - ‘intro_stat’, ‘STAT’, ‘statview’ and ‘statgui’ man pages
Hung application with STAT

% qstat -f 722272

Find the MOM node that launched the app.

... login_node_id = nid02051

% ssh -XY nid02051

Log into the MOM node

% ps -f

Find pid

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>wyang</td>
<td>23953</td>
<td>16045</td>
<td>0</td>
<td>Feb01</td>
<td>pts</td>
<td>00:00:00</td>
<td>aprun -n 4 ./jacobi_mpi</td>
</tr>
<tr>
<td>wyang</td>
<td>23961</td>
<td>23953</td>
<td>0</td>
<td>Feb01</td>
<td>pts</td>
<td>00:00:00</td>
<td>aprun -n 4 ./jacobi_mpi</td>
</tr>
</tbody>
</table>

... % module load stat

% stat -i 23953

Run ‘stat’ for the process 10921; -i to get source line numbers

STAT samples stack backtraces a few times

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 /scratch1/scratchdirs/wyang/parallel_jacobi/stat_results/jacobi_mpi.0010

% ls -l stat_results/jacobi_mpi.0010/*.dot
-rw-r----- 1 wyang wyang 2227 Feb 2 00:09 stat_results/jacobi_mpi.0010/jacobi_mpi.0010.3D.dot

% statview stat_results/jacobi_mpi.0010/jacobi_mpi.0010.3D.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 of a failing application
  – Invokes STAT underneath
  – Output in atpMergedBT.dot and atpMergedBT_line.dot (which shows source code
    line numbers), which are to be viewed with statview

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

  Include this in your dot file (e.g., .tcshrc.ext) to enable ATP by default

• Can make core dumps (core.atp.apid.rank), too, by setting coredumpsize
  unlimited:
  – unlimit coredumpsize # for csh/tcsh
  – ulimit -c unlimited # for sh/bash/ksh

  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
Running an application with ATP

```bash
% cp $ATP_HOME/demos/testMPIApp.c .
% cc -o testMPIApp testMPIApp.c
% cat runit
#!/bin/csh
#PBS -l mppwidth=24
#PBS -l walltime=5:00
#PBS -q debug
#PBS -j oe

cd $PBS_O_WORKDIR
setenv ATP_ENABLED 1
aprun -n 8 ./testMPIApp 1 4

% qsub runit
714152.edique02

% cat runit.o714152
...
Application 2885291 is crashing. ATP analysis proceeding...
...
Process died with signal 4: 'Illegal instruction'
View application merged backtrace tree with: statview atpMergedBT.dot
...
% module load stat
% statview atpMergedBT.dot  # or statview atpMergedBT_line.dot
```
Hung application with ATP

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

% apstat Find apid
...
Apid ResId User PEs Nodes Age State Command
2885161 140092 wyang 4 1 0h02m run jacobi_mpi
...
% apkill 2885161 Kill the application
% cat runit.o714080
...
aprun: Apid 2885161: Caught signal Terminated, sending to application
...
Process died with signal 15: 'Terminated'
View application merged backtrace tree with: statview atpMergedBT.dot
...
% module load stat
% statview atpMergedBT.dot # or statview atpMergedBT_line.dot
CCDB (Cray Comparative Debugger)

• Find a bug introduced in a version, by running two versions side by side and comparing data between them
• GUI
• It runs the command line mode version, lgdb (Cray Line Mode Parallel Debugger), underneath
• Supports MPI; doesn’t support threading
• For info:
  – ccdb man page and help pages
  – lgdb man page and help pages
  – ‘Using the lgdb Comparative Debugging Feature’,
CCDB (Cont’d)

• To compare something between two applications, need to specify
  – Variable name
  – Location in a source file
  – How the global data for the variable is distributed over MPI processes
  – Set of MPI processes (“PE set”) for the distribution

• 3 entities used in CCDB (and lgdb)
  – PE set: A set of MPI processes
  – Decomposition: How an array is distributed over PEs
  – Assertion script: A collection of mathematical relationships (e.g., equality) to be tested
Running CCDB

```bash
% qsub -IV -lmppwidth=48,walltime=30:00 -q debug
% cd $PBS_O_WORKDIR
% module load cray-ccdb
% ccdb
```

Request enough nodes to run two apps. simultaneously.
CCDB assertion script

- This script tests whether the 6 variables have the same values between the applications, at line 418 of HPL_pdtest.c; resid0 and Xnorml don’t.
Valgrind

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

• For info:
Valgrind’s memcheck

```bash
% qsub -IV -lmppwidth=24,walltime=30:00 -q debug
% cd $PBS_O_WORKDIR
% module load valgrind
% ftn -dynamic -g -O0 memory_leaks.f $VALGRIND_MPI_LINK
% aprun -n 2 valgrind --leak-check=full ./a.out >& report
% awk ‘/^==/ {print $1}’ report | sort -u

==46886==
==46887==
==46888==
==46889==

• Out of 4 sub-reports, two are for the application’s 2 MPI tasks; let’s look at the one for process ID 46888

% awk ‘/^==46888/ {print}’ report

... ==46888== 8,000,000 bytes in 2 blocks are possibly lost in loss record 33 of 37
==46888== at 0x4C27F9E: malloc (vg_replace_malloc.c:291)
==46888== by 0x424A93: for_allocate (in /scratch1/scratchdirs/wyang/valgrind/a.out)
==46888== by 0x408269: sub_bad_ (memory_leaks.f:37)
==46888== by 0x407DF5: MAIN__ (memory_leaks.f:14)
==46888== by 0x407D55: main (in /scratch1/scratchdirs/wyang/valgrind/a.out)

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

% ftn -g -O2 memory_leaks.f
% aprun -n 2 valgrind --tool=cachegrind ./a.out
% ls -lrt

• It generates 4 separate reports just like before, and two are for the application’s 2 MPI tasks; let’s look at the one for process ID 46849

% cg_annotate cachegrind.out.46849

---------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         31457280 B, 64 B, 30-way associative
Command:          ./a.out
Data file:        cachegrind.out.46849
---------------------------------------------------------------------------------
<table>
<thead>
<tr>
<th>Ir</th>
<th>Ilmr</th>
<th>ILmr</th>
<th>Dr</th>
<th>D1mr</th>
<th>DLmr</th>
<th>Dw</th>
<th>D1mw</th>
<th>DLmw</th>
<th>file: function</th>
</tr>
</thead>
<tbody>
<tr>
<td>220,000,000</td>
<td>4</td>
<td>12</td>
<td>20,000,000</td>
<td>0</td>
<td>0</td>
<td>???:for_random_number_single</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25,250,536</td>
<td>43</td>
<td>40</td>
<td>8,000,073</td>
<td>250,020</td>
<td>187,498</td>
<td>/some/path/ memory_leak.f:MAIN__</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Valgrind’s callgrind

```sh
% ftn -g -O2 memory_leaks.f
% aprun -n 2 valgrind --tool=callgrind ./a.out
% ls -lrt
...
-rw------- 1 wyang wyang 38527 Feb 1 13:16 callgrind.out.43223
-rw------- 1 wyang wyang 117581 Feb 1 13:17 callgrind.out.43225
-rw------- 1 wyang wyang 124967 Feb 1 13:17 callgrind.out.43224
-rw------- 1 wyang wyang 47409 Feb 1 13:17 callgrind.out.43222
```

- It generates 4 separate reports just like before, and two are for the application’s 2 MPI tasks; let’s look at the one for process ID 43224

```sh
% callgrind_annotate callgrind.out.43224 memory_leaks.f
```

---

---

-- User-annotated source: memory_leaks.f

---

```fortran
subroutine sub_ok(val,n)      ! no memory leak
  integer n
  real val
  40 real, allocatable :: a(:)
 240 allocate (a(n))
 2,425 => ???:for_alloc_allocatable (10x)
 810 => ???:for_check_mult_overflow64 (10x)
 50,000,070 call random_number(a)
 550,000,000 => ???:for_random_number_single (10000000x)
 3,125,926 val = val + sum(a)
 180 end
```

Use -g but keep the usual optimization level.
Valgrind’s heap profilers

- massif and exp-dhat for profiling heap memory usage

% ifort -g -O2 memoryLeaks_serial.f
% valgrind --tool=massif ./a.out
% ls -lrt
...
-rw------- 1 wyang wyang 12008 Feb 1 12:40 massif.out.1384
% ms_print massif.out.1384
...

<table>
<thead>
<tr>
<th>MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>114.5^</td>
</tr>
</tbody>
</table>

@ for detailed snapshot where detailed info is provided

# for peak snapshot where the peak heap usage is

This example strongly suggests memory leaks

<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>67</td>
<td>2,414,128,664</td>
<td>120,000,280</td>
<td>120,000,032</td>
<td>248</td>
<td>0</td>
</tr>
</tbody>
</table>
Thank you.