Cray Debugging Support Tools

Luiz DeRose
Sr. Principal Engineer
Programming Environments Director
Cray Inc.
Debugging on Cray Systems

- **Systems with thousands of threads of execution need a new debugging paradigm**

- **Cray’s focus is to build tools around traditional debuggers with innovative techniques for productivity and scalability**
  - Support for traditional debugging mechanism
    - RogueWave TotalView and Allinea DDT
  - Scalable Solutions based on MRNet from University of Wisconsin
    - **STAT - Stack Trace Analysis Tool**
      - Scalable generation of a single, merged, stack backtrace tree
    - **ATP - Abnormal Termination Processing**
      - Scalable analysis of a sick application, delivering a STAT tree and a minimal, comprehensive, core file set.
  - **lgdb 2.0**
    - Ability to see data from multiple processors in the same instance of lgdb
      - without the need for multiple windows
    - **Comparative debugging**
      - A data-centric paradigm instead of the traditional control-centric paradigm
      - Collaboration with Monash University and University of Wisconsin for scalability
  - **Fast Track Debugging**
    - Debugging optimized applications
    - Added to Allinea’s DDT 2.6 (June 2010)
MRNet - Multicast Reduction Network

- Tree based software overlay network

- Provides efficient multicast and reduction communications for parallel and distributed tools

- Uses a tree of processes between the tool's front-end and back-ends to improve group communication performance
  - Internal processes are used to distribute important tool activities
    - Reduce data analysis time
    - Keep tool front-end loads manageable
Stack Trace Analysis Tool (STAT)

● Stack trace sampling and analysis for large scale applications
  ● Sample application stack traces
  ● Scalable generation of a single, merged, stack backtrace tree
    ● A comprehensible view of the entire application
    ● Discover equivalent process behavior
      ● Group similar processes
      ● Reduce number of tasks to debug
    ● 128K processes analyzed in 2.7 seconds, using MRNet

● Merge/analyze traces:
  ● Facilitate scalable analysis/data presentation
  ● Multiple traces over space or time
  ● Create call graph prefix tree
    ● Compressed representation
    ● Scalable visualization
    ● Scalable analysis
Stack Trace Merge Example

_start

__libc_start_main

main

func1

func2

func3

func4

_start

__libc_start_main

main

func1

func2

func3

func4

func5

func6

_start

__libc_start_main

main

func1

func2

func5

func6
2D-Trace/Space Analysis
STATview & STATGUI

- **STATview** is a GUI for viewing STAT outputted DOT files
  - STATview provides easy navigation of the call prefix tree and also allows manipulation of the call tree to help focus on areas of interest

- **STATGUI** is a GUI that drives STAT and allows you to interactively control the sampling of stack traces from your parallel application
  - STATGUI is built on top of STATview and provides the same call tree manipulation operations
  - In addition to the operations provided by STATview, STATGUI provides a toolbar to control STAT's operations

- **STATGUI** can also serve as an interface to attach a full-featured debugger such as DDT to a subset of the application tasks
STAT 1.2.1.3

- **module load stat**
  - Not loaded by default

- **man STAT**

- **STAT <pid_of_aprun>**
  - Creates STAT_results/<app_name>/<merged_bt_file>

- **Scaling limited by number file descriptor**
When a large scale parallel application dies, one, many, or all processes might trap!
- It is next to impossible to examine all the core files and backtraces
  - No one wants that many stack backtraces
  - No one wants that many core files
    - They are too slow and too big
    - Sufficient storage for all core files is a problem
  - They are too much to comprehend
- A single core file or stack backtrace is usually not enough to debug either!
  - A single backtrace produced might not be from the process that first failed

Requirements:
- Minimum jitter
- Scalability
- Robustness
- Small footprint
- Limited core file dumping

ATP 1.6.1 was released in January 2013
ATP Description

- System of light weight back-end monitor processes on compute nodes
  - Coupled together with MRNet
  - Automatically launched by aprun in parallel with application launch
    - Enabled/disabled via ATP_ENABLED environment variable

- Leap into action on any application process trapping
  - stderr backtrace of first process to trap
    - dumps core file set (if limit/ulimit allows)
  - Uses StackwalkerAPI to collect individual stack backtraces, even for optimized code

- STAT like analysis provides merged stack backtrace tree
  - Leaf nodes of tree define a modest set of processes to core dump
    - or, a set of processes to attach to with a debugger
ATP produces a single merged stack trace

- or a **reduced set of core files**
  - ATP selects a single representative from each leaf node of the merged stack backtrace tree
  - Each core file is named core.atp.apid.rank
  - Users can control, to some degree, the set of core dumps created by ATP

**The benefits:**

- Minimal impact on application run
  - Can be used with production runs
- Automated, transparent collection of data
- Ability to hold failing application for close inspection
  - This is site dependent
- Easy to navigate the merged stack trace
- Manageable set of core files
- Reduced amount of data saved
  - Especially true in the core file situation
ATP: How It Works

- ATP is launched via an ALPS enhancement which includes the fork/exec of a login side ATP front-end daemon
  - The ATP front-end uses MRNet and the ALPS tool helper library to launch ATP back-end servers on all compute nodes associated with the application

- ATP signal handler runs within an application to catch fatal errors
  - It handles the following signals:
    - SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIGXCPU, SIGXFSZ
  - Setting the environment variables MPICH_ABORT_ON_ERROR and SHMEM_ABORT_ON_ERROR will cause a signal to be thrown and captured for MPI and SHMEM fatal errors

- ATP daemon running on the compute node captures signals, starts termination processing
  - Rest of the application processes are notified
  - Generates a stacktrace
  - Creates a single merged stack trace file

- The stack trace file is viewed with the STATview tool
ATP Hold Time

- ATP is able to hold a dying application in stasis in order to allow the user to attach to it with a debugger
  - To do so, set the `ATP_HOLD_TIME` environment variable to the number of minutes desired

- Once attached, the debugging session can last as long as the batch system allows
  - Which in turn depends on the compute node resources you requested when you began your session
  - So use ATP_HOLD_TIME to define the time you need to attach to the application, not the total time needed for the debugging session.

- If ATP_HOLD_TIME is set, core dumping is disabled
Comparative Debugger

● Collaboration with Monash University
  ● A data-centric paradigm instead of the traditional control-centric paradigm

● Helps the programmer locate errors in the program by observing the divergence in key data structures as the programs are executing

  ● Allows comparison of a “suspect” program against a “reference” code using assertions
    ● Simultaneous execution of both
    ● Ability to assert the match of data at given points in execution
    ● Focus on data – not state and internal operations
    ● Narrow down problem without massive thread study

● Data comparison
  ● Tolerance control – nobody expect it to be perfect
  ● Array subsets – correlate serial to parallel bits
  ● Array index permutation – loops rearranged
  ● Automated asserts – let it run until a problem is found
  ● Forcing correct values – continue on with correct data
Need a way to declare that we expect two pieces of data are equivalent
  - Backup: What is data?
  - Define specific variables in the source (where?)
  - Define a particular line number to observe the variables (when?)

Assertions provide this ability
  - Assert that the two should be equivalent at that moment
Assertions, Graphs and Blockmaps…

Oh my!

- But wait, there's more
  - Want to compare multiple variables at the same line number in the code
  - Want to compare a single variable at many different line numbers in the code.

- There’s a graph for that
  - Execute many different assertions simultaneously
Assertions, Graphs and Blockmaps…

Oh my!

● Sometimes assertions alone are not enough
  ● Serial data to distributed data
  ● One-dimensional to multi-dimensional data.
  ● Scalar to non-scalar data.

● Blockmaps provide a simple mechanism to decompose data
  ● Based on HPF syntax
  ● Allows for block, cyclic, and * (wildcard) decomposition definitions
  ● Defines how the data is distributed across a set of parallel processes

Example of a (block,*) decomposition.
Putting It Together

- A graph is made up of assertions which contains data definitions
  - Data versus data, blockmap versus data, blockmap versus blockmap.
- Once defined, a graph is executed
Comparative Debugger Status

- Released with lgdb 2.0.0 (November 2012)
  - module load cray-lgdb
    - On Blue Waters - cray-lgdb/2.0.1(default)

- Supports applications compiled with CCE, PGI, and GNU Fortran, C, and C++ compilers.

- Basic operation is documented in the lgdb man page
  - man lgdb(1)

- A white paper on “Using the lgdb Comparative Debugging Feature” will be available soon

- We are working on a graphical user interface (GUI) for better ease of use
Fast Track Debugging: The Problem Being Solved

- How to debug parallel optimized codes

- **Debug flags eliminate optimizations**
  - Today's machines really need optimizations
  - Slows down execution
  - Problem might disappear

- **Fast Track Debugging addresses this problem**
How to do "Fast Track Debugging"?

● Compile such that both debug and non-debug (optimized) versions of each routine are created
  ● Debug and non-debug versions of each subroutine appear in the executable

● Linkage such that optimized versions are used by default

● User sets breakpoints or other debug constructs
  ● Debugger overrides default linkage when setting breakpoints and stepping into functions
  ● Routines automatically presented using the debug version of the routine
  ● Rest of program executes using optimized versions of the routines
A Closer Look at How FTD Works

source code

---
call difuze(...)---
call interf(...)---

subroutine difuze(...)---
---

optimized binary code

difuze()

call difuze(...)---
call interf(...)---
difuze debug()

subroutine interf(...)---
---

debug code

interf()

call interf(...)---
call interf(...)---
interf debug()

---

Jmp inserted as part of breakpoint planting

Breakpoint requested in interf(), placed in interf_debug()
Tera TF Execution Time
Fast Track Debugger – Issues / Cost

- Compiles are slower
- Executable uses more disk space
- Libraries probably don't have a debug version
- Inlining turned off
  - 1.7% average slow down of all SPEC2007MPI tests
  - Range of slight speedup to 19.5% slow down
- Uses more memory
  - 4% larger at start up
  - 0.0001% larger after computation
Fast Track Debugger Status

- Support available in the Cray Compilation Environment (CCE)

- Prototype in gdb
  - Exercised through lgdb

- Added to Allinea's DDT 2.6 (June 2010)