CRAY
THE SUPERCOMPUTER COMPANY

Cray Debugging Support for Large Scale Systems

Luiz DeRose
Programming Environments Director
Cray Inc.

Acknowledgements

- 

- Paradyn Group – University of Wisconsin
- Future Technologies Group / ORNL
- ADEPT / LLNL
- Monash University
- Allinea

The Next Generation of Debuggers on Cray Systems

- Peta-scale systems with hundreds of thousands of threads need a new debugging paradigm
  - Innovative techniques for productivity and scalability
    - Scalable Solutions based on MRNet
      - 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.
    - Comparative debugging
      - A **data-centric paradigm** instead of the traditional control-centric paradigm
    - Fast Track Debugging
      - Debugging optimized applications
  - Support for traditional debugging mechanism
    - TotalView, DDT, and gdb

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

MRNet Based Debugging Support Tools

- STAT - Stack Trace Analysis Tool
  - Scalable generation of a single, merged, stack backtrace tree
    - 128K processes analyzed in 2.7 seconds, using MRNet
    - A comprehensible view of the entire application
    - Focuses and directs debugger subset attach opportunities
- ATP - Abnormal Termination Processing
  - Scalable analysis of a sick application, delivering a STAT tree and a minimal, comprehensive, core file set
    - Minimal impact on application run
    - Automated, transparent collection of data
    - Ability to hold failing application for close inspection

ATP: The Problem Being Solved

- Applications on Cray systems use hundreds of thousands of processes

- 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

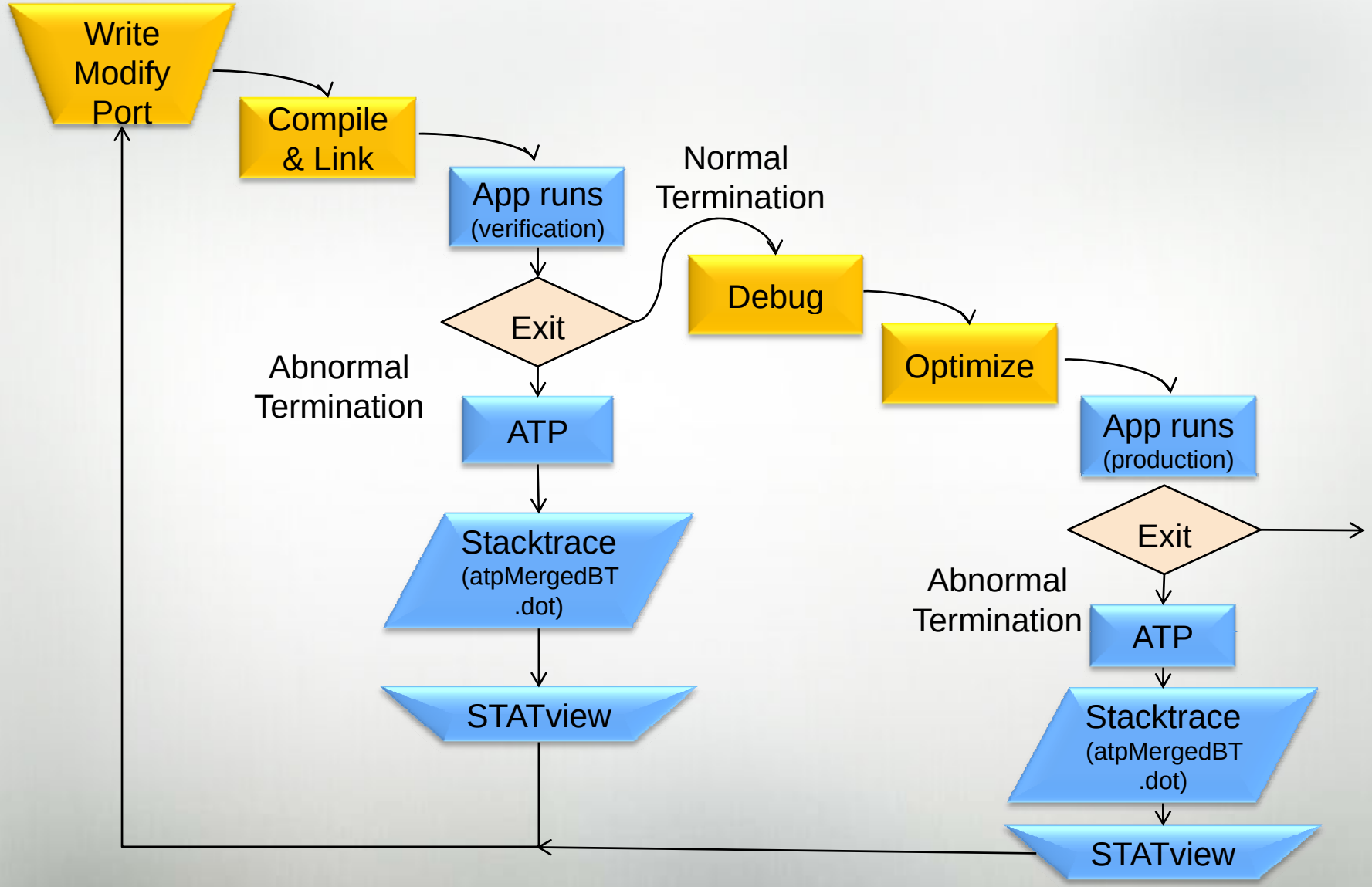
Abnormal Termination Processing

- ATP produces a single merged stack trace
  - or a reduced set of core files (future work)
- The benefits:
  - Easy to navigate the merged stack trace
  - Manageable set of core files
  - Reduced amount of data saved
    - Especially true in the core file situation
- Requirements:
  - Minimum jitter
  - Scalability
  - Robustness
  - Small footprint
  - Limited core file dumping

ATP Description

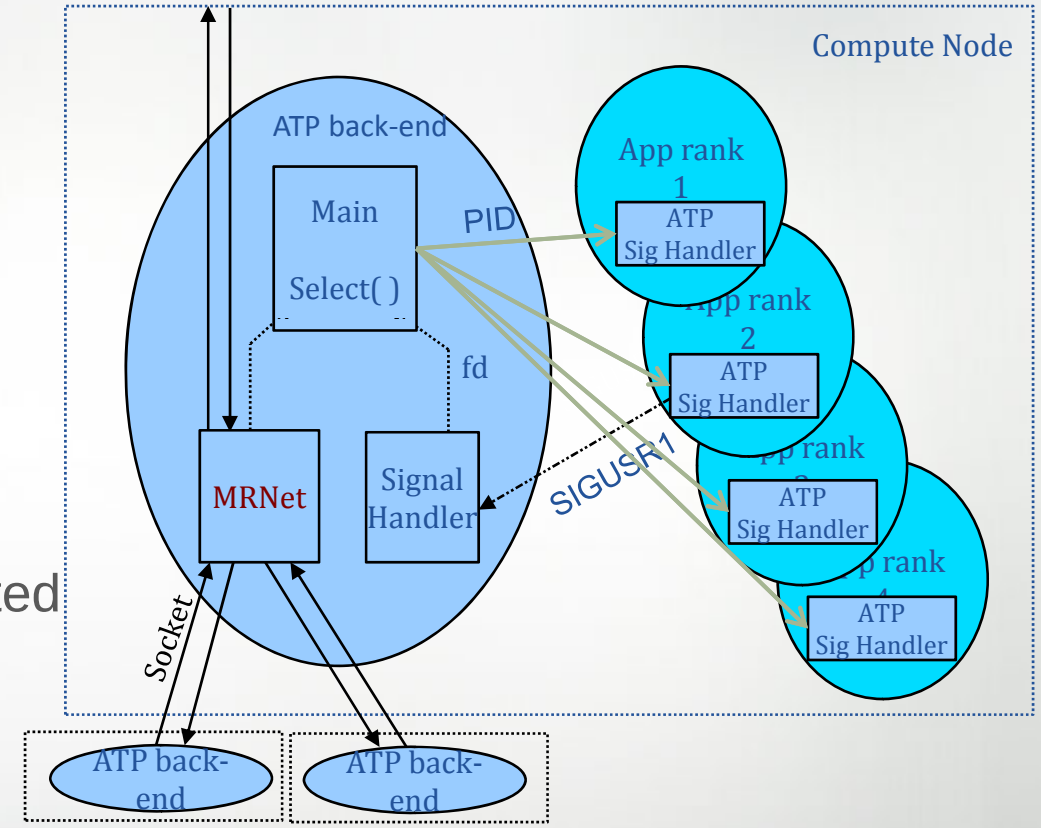
- System of light weight back-end monitor processes on compute nodes
  - Coupled together with **MRNet**
- Leap into action on any application process trapping
- **STAT** like analysis provides merged stack backtrace tree
  - Leaf nodes of tree define a modest set of processes to core dump
    - or, a set of processes to attach to with a debugger
- Uses **StackwalkerAPI** to collect individual stack backtraces, even for optimized code

ATP – Abnormal Termination Processing



ATP Components

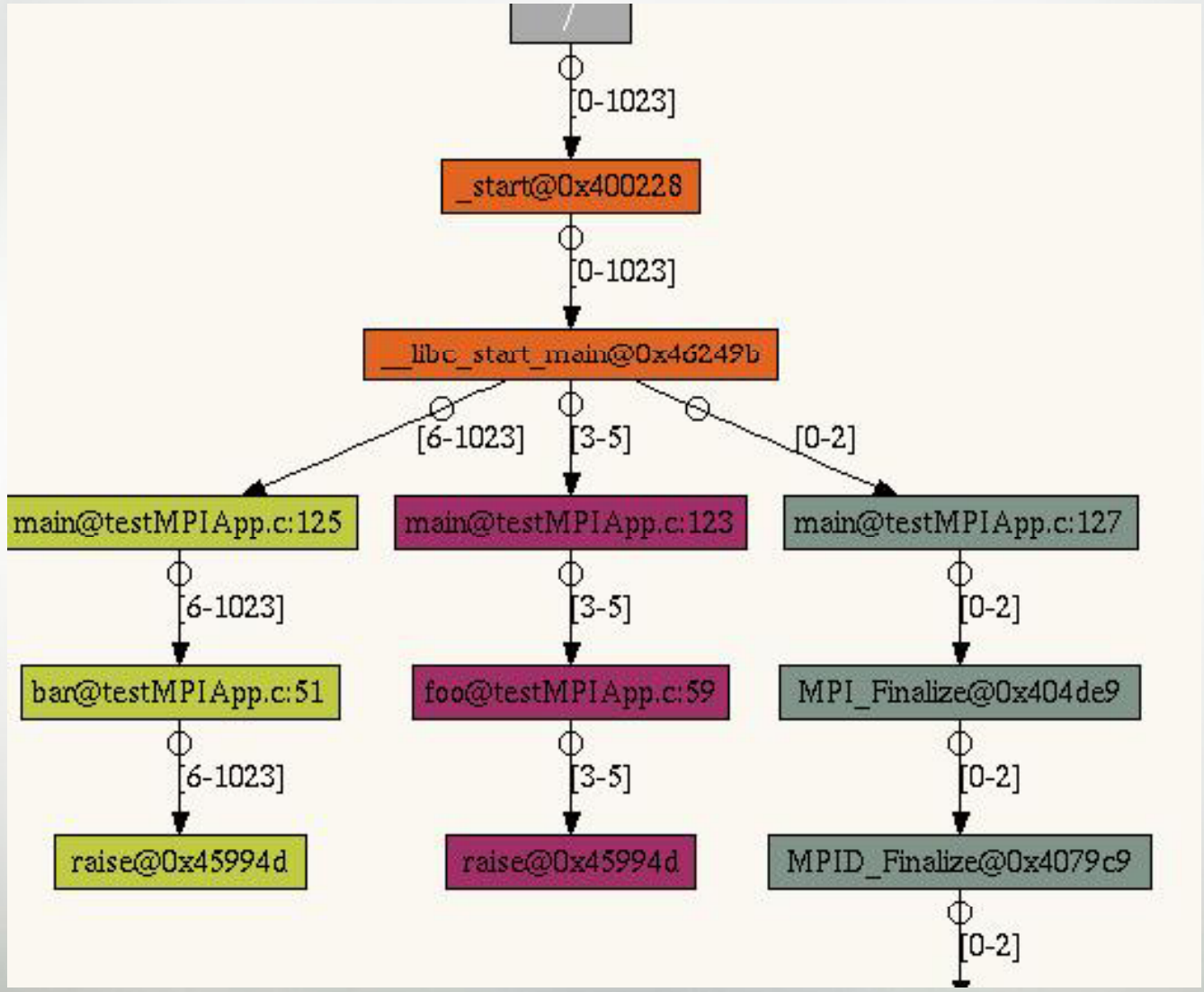
- Application process signal handler
  - Triggers analysis
  - Controls its own RLIMIT_CORE and core_pattern
- Back-end monitor
  - Collects backtraces via StackwalkerAPI
  - Forces core dumps as directed
- Front-end controller
  - Coordinates analysis via MRNet
  - Selects process set that is to dump core



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

Merged Stacktrace Tree Example



ATP 1.0.2

- Released May 20, 2010
- Manual (rather than automatic)
  - Must add code to application
  - Must link specially
  - Must explicitly launch (CLE 2.2)
- Provide backtrace of first crash to stderr
- Provides merged backtrace tree
- Tested at 1024 PEs

ATP – Future Work

- Automatic invocation of ATP
  - Today users need to insert signal handler
  - With next release of OS, just need to load atp module
- Hold a dying application in stasis
  - Gives the user an opportunity to attach a debugger to the application
- Send email notification to user that job has failed
- Improved scalability
  - ATP stack backtraces have been produced on applications made up of about 2000 processes
  - Expect to be able to handle applications with 100,000s of processes in the future

Data Centric Debugging

- Direct control of millions of threads is unworkable
  - Need to narrow down to a limited number of threads
  - Direct control of limited threads, once narrowed down
- Narrowing down requires a different approach
  - **Data-centric view**, instead of the traditional control flow view
    - In a data-centric view users make statements about the contents of the data structures and the veracity of these are checked by the debugger

What is a comparative debugger?

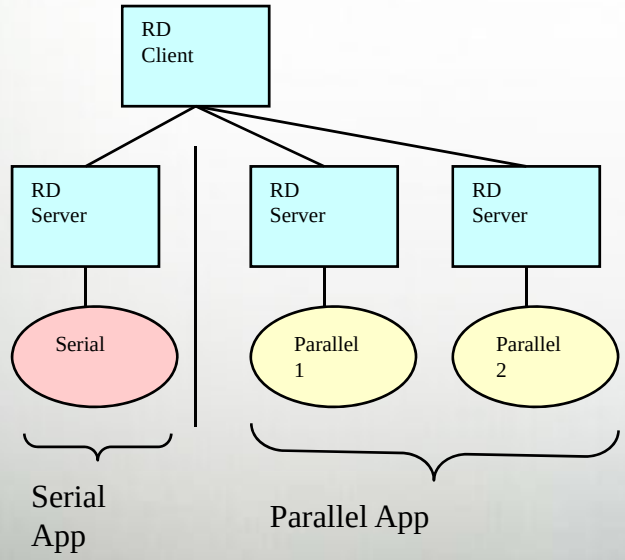
- Originally from Monash University in Australia
- Helps the programmer locate errors in the program by observing the divergence in key data structures as the programs are executing
- Allows comparison of a “suspect” program against a “reference” code using assertions
  - Simultaneous execution of both
  - 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 centric – no additional complexity as scale increases
- Cray and Monash University are working together under a grant from the Australian government on scalability research.

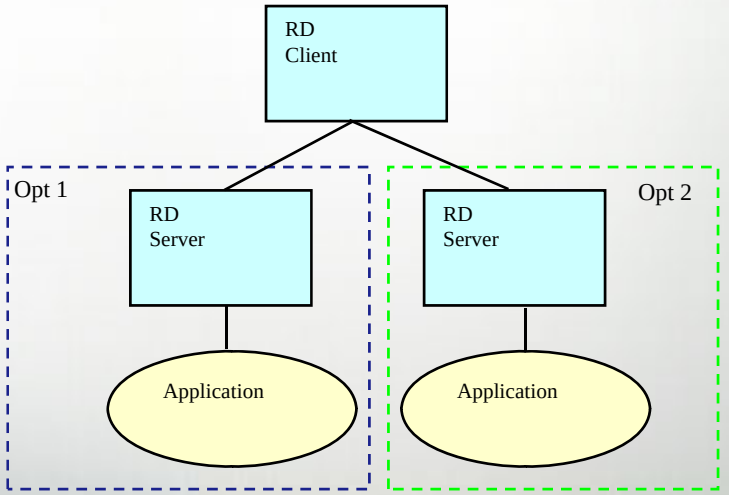
Assertions

- Data assertions are the heart of comparative debugging
  - Assert that data1 at location1 matches data2 at location2
- The comparative debugger verifies this assertion as the applications execute
- When the assertion fails, one has a specific region of the failing application to inspect
  - This often means that the user iterates with a refined assertion to further narrow the search area
- Assertions can be simple (scalar to scalar) or more complex (serial to parallel multidimensional arrays)

Comparative Debugging Scenarios

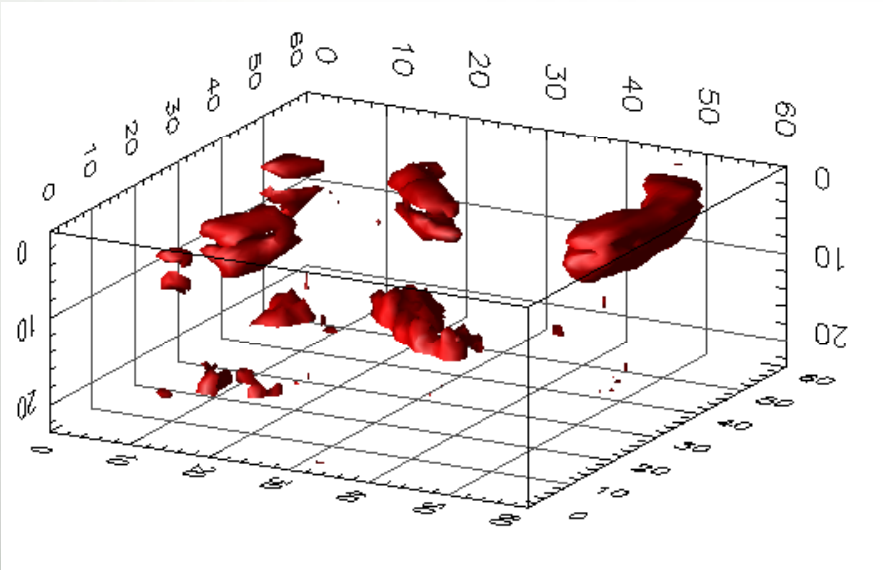
- Serial converted to parallel
- Small scaling versus large scaling
- One optimization level versus another
- One programming language converted to another
- . . .

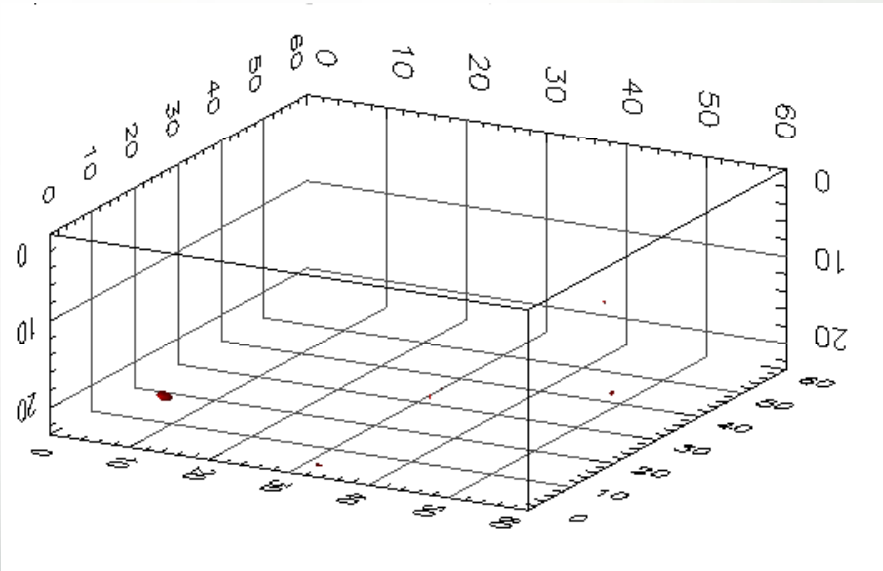




Comparative Debugging Example

- MM5 Weather model
- Serial versus parallel
- Difference >0.1%
- Narrowed to missing term in equation
- **… term found and added in**





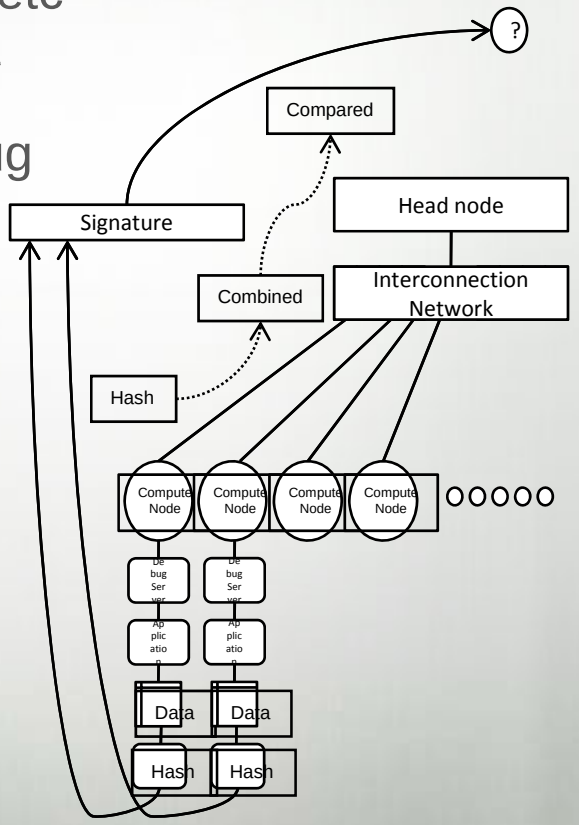
Courtesy of David Abramson from Monash University

Cray Comparative Debugging at Scale

- Existing implementation of comparative debugger running on Cray XT and XE Systems

- Modifying to improve scalability
  - Add process sets as primitive data types
  - Implement set operations for setting breakpoints, etc
  - Considering MRNET tree for gathering breakpoint events
  - Removing reliance on socket connections to debug servers
  - Simplified original data decomposition specification language to match current high level languages

- Experimenting with alternative comparison techniques
  - Computing signatures on decomposed data structures rather than always importing data for comparison



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

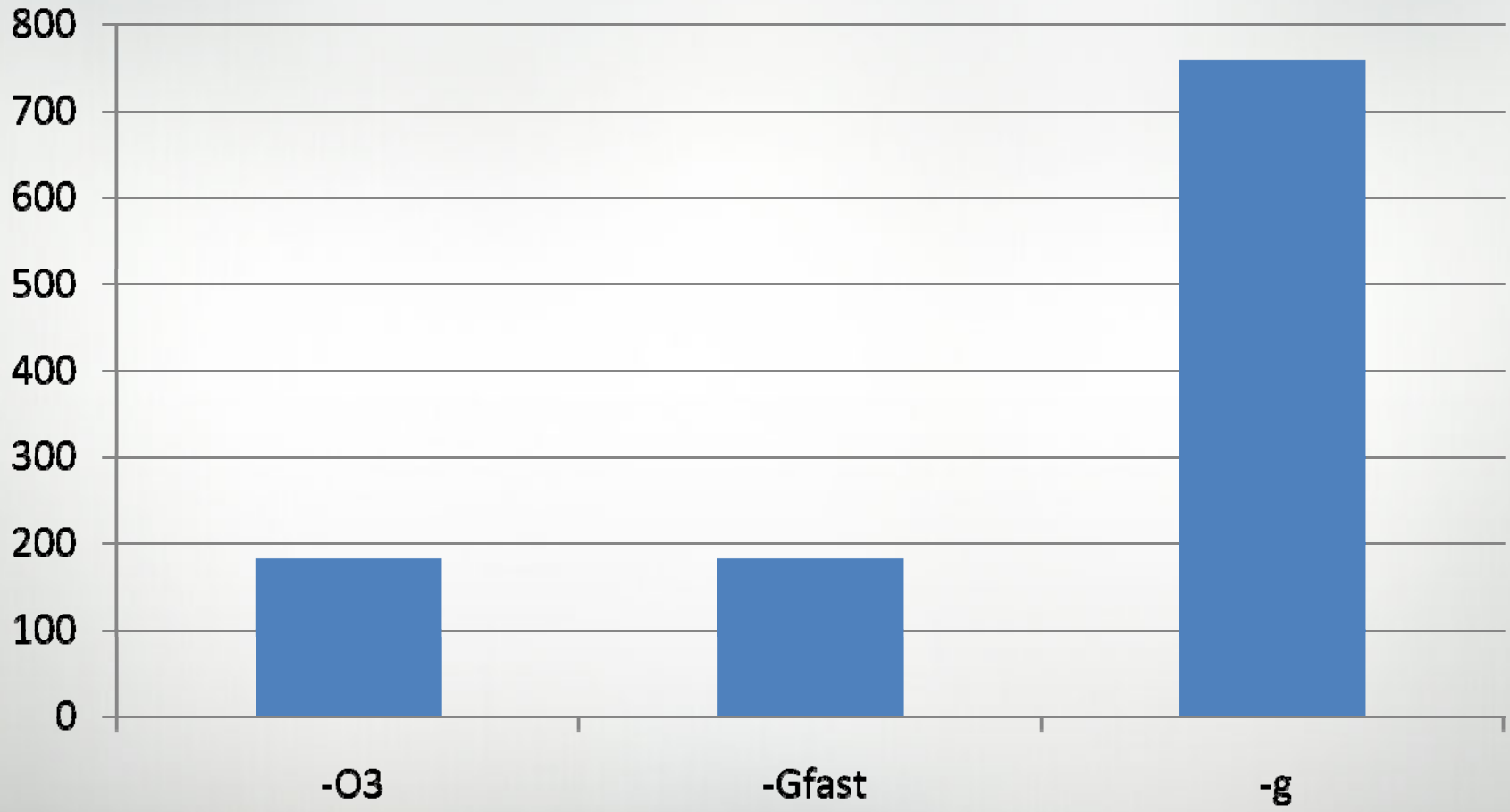
How to do "Fast Track Debugging"?

- 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

Fast Track Debugger Status

- Support available in the Cray Compiling Environment
- Prototype in gdb
  - Exercised through lgdb
- Added to Allinea's DDT 2.6 (June 2010)

Tera TF Execution Time



Fast Track Debugging: Issues / Cost

- Compiles are slower
- Executable uses more disk space
- 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

Luiz DeRose (ldr@cray.com)