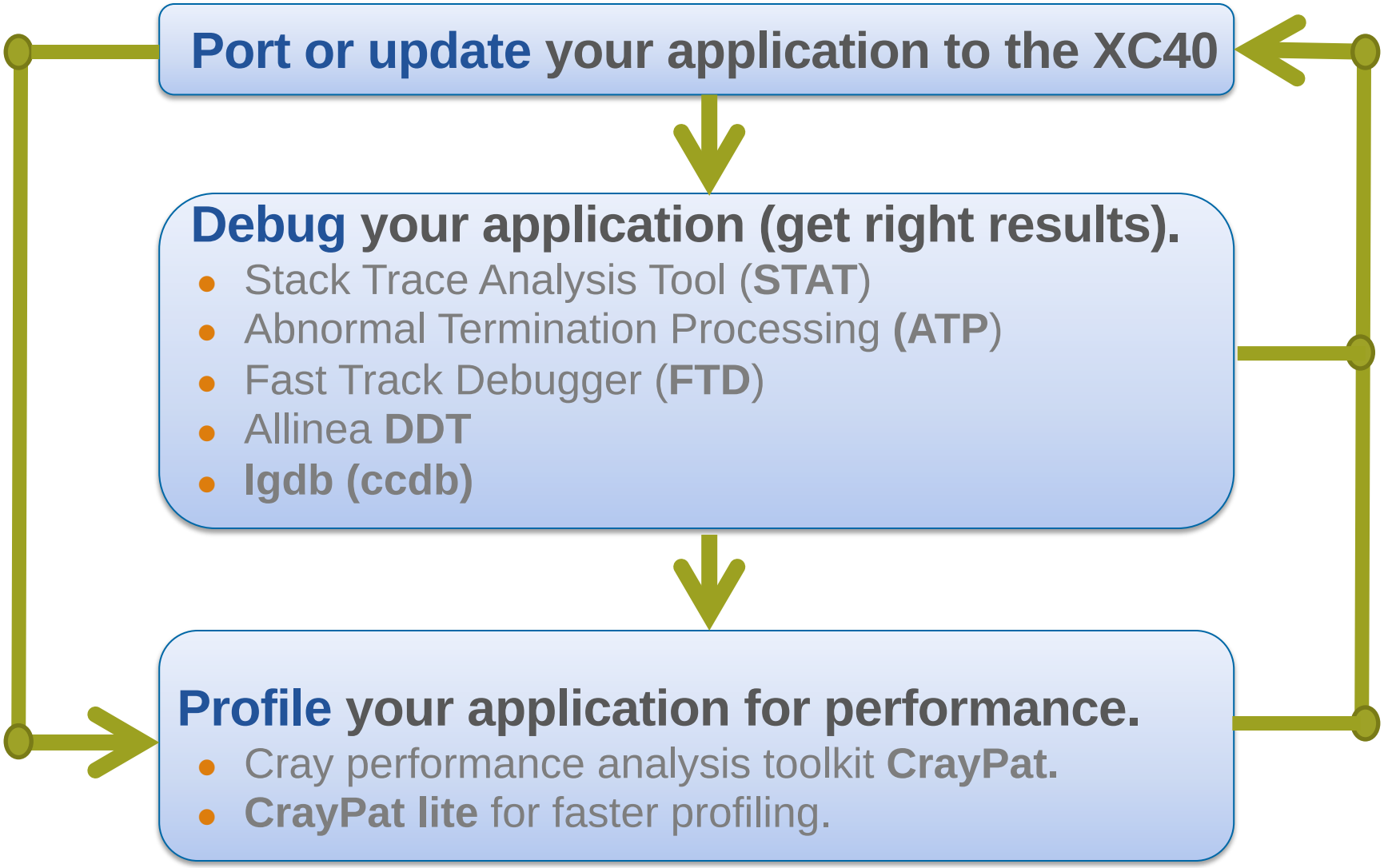
Short Introduction to Debugging Tools on the Cray XC40

Overview

	Lightweight At most relinking. Get a first picture of a performance or problems during execution.	**In-depth** Recompile/Relink. Provides detailed information at user routine level.
Debugging Get your code up and running correctly.	**ATP** **STAT**	**lgdb, ccdb** **Fast track** **Allinea DDT**
Profiling Locate performance bottlenecks.	**CrayPAT-lite** **IOBUF**	**CrayPAT** **Apprentice2** **Reveal**

- **Some of the above are explained in separate presentations. More information about Cray Tools in man pages and on docs.cray.com**

The porting optimization Cycle



Debugging in production and scale

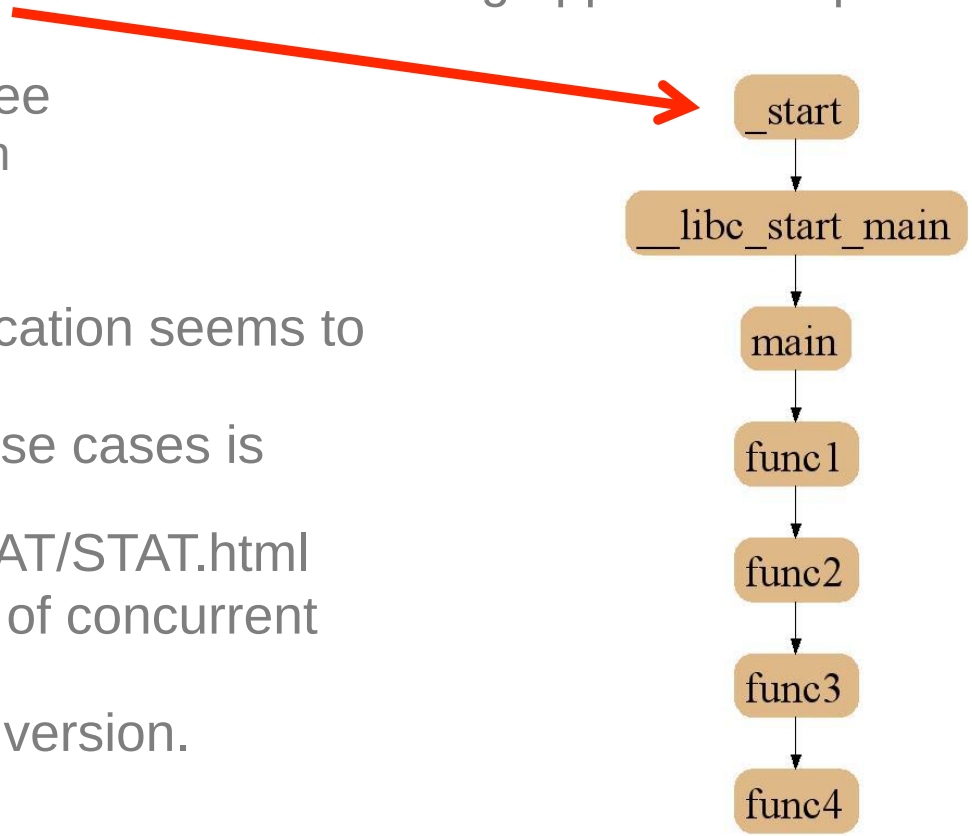
- **Even with the most rigorous testing, bugs may occur during development or production runs.**
  - It can be very difficult to recreate a crash without additional information
  - Even worse, for production codes need to be efficient so usually have debugging disabled
- **The failing application may have been using tens of or hundreds of thousands of processes**
  - If a crash occurs one, many, or all of the processes might issue a signal.
  - We don't want the core files from every crashed process, they're slow and too big!
  - We don't want a backtrace from every process, they're difficult to comprehend and analyze.

Stack Trace Analysis Tool (STAT)

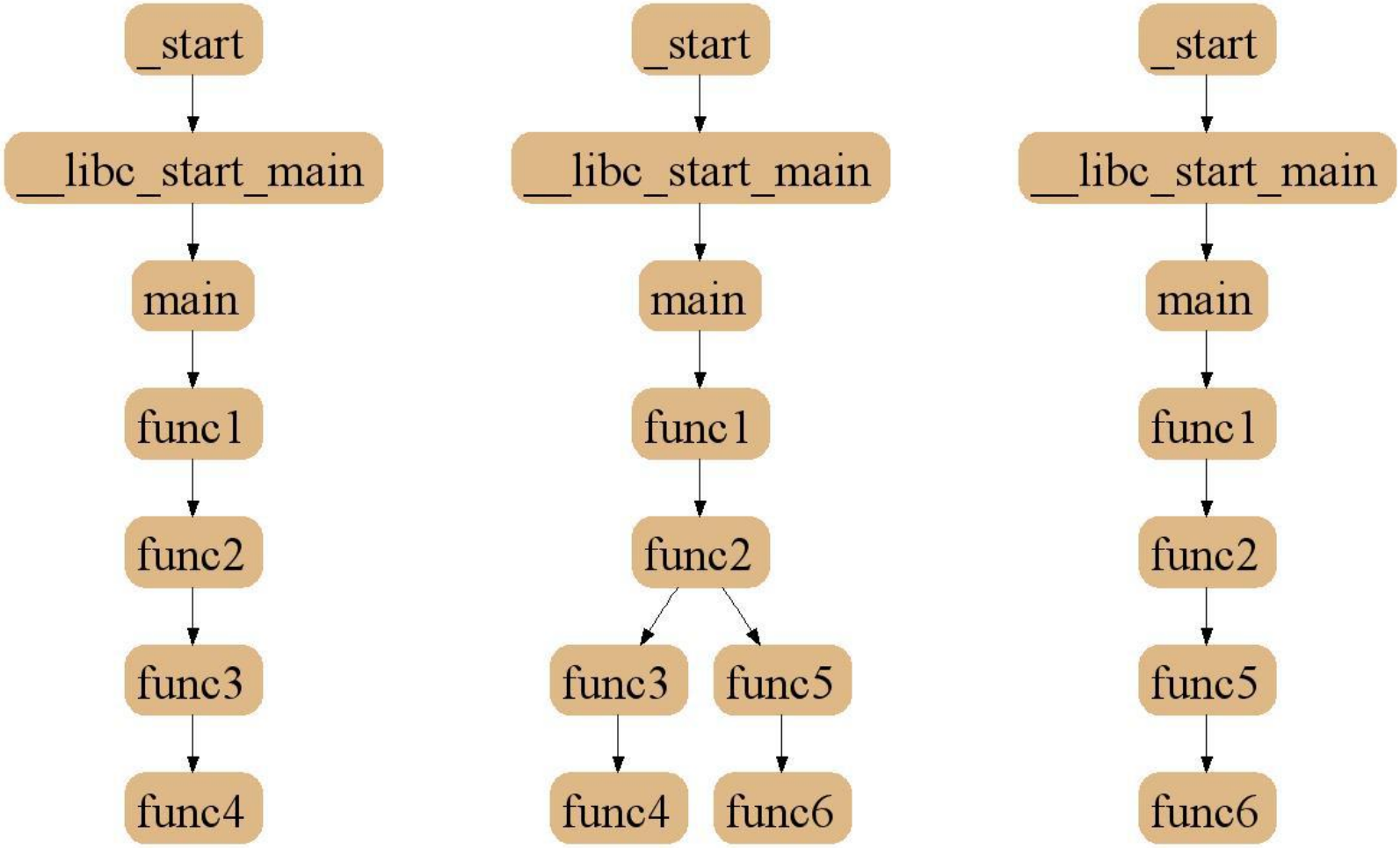
For when nothing appears to be happening…

Stack Trace Analysis Tool (STAT)

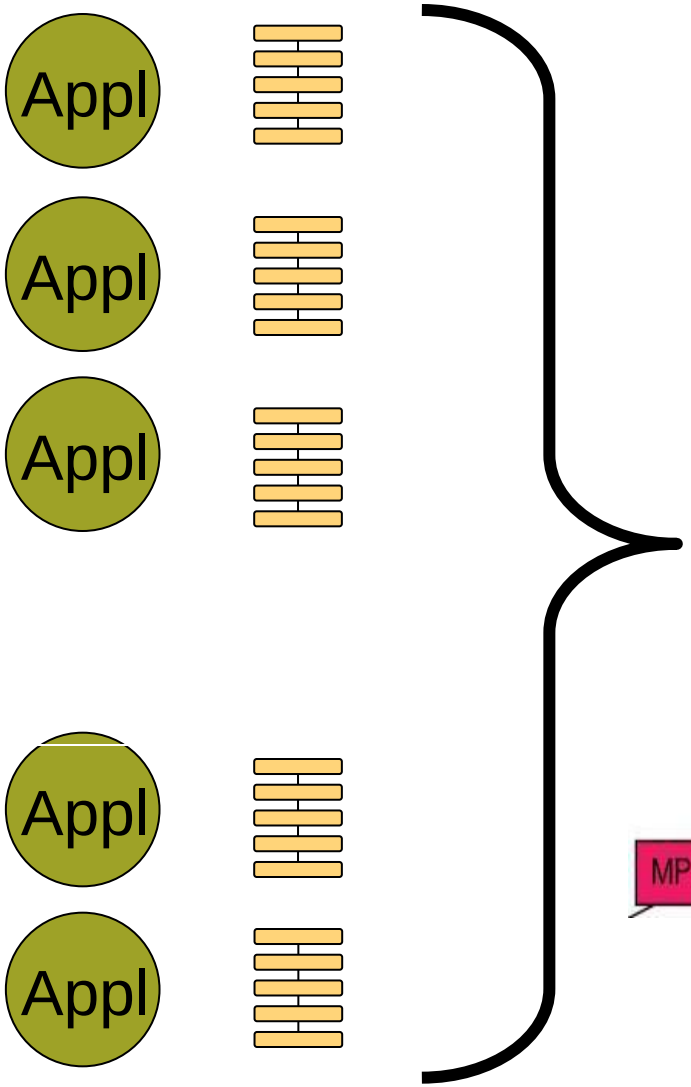
- **Stack Trace Analysis Tool (STAT) is a cross-platform tool from the University of Wisconsin-Madison.**
  - Gathers and merges stack traces from a running application's parallel processes.
  - Creates call graph prefix tree
    - Compressed representation
    - Scalable visualization
    - Scalable analysis
  - It is very useful when application seems to be stuck/hung
  - Full information including use cases is available at http://www.paradyn.org/STAT/STAT.html
  - Scales to many thousands of concurrent process.
  - STAT 2.1.0.1 is the default version.

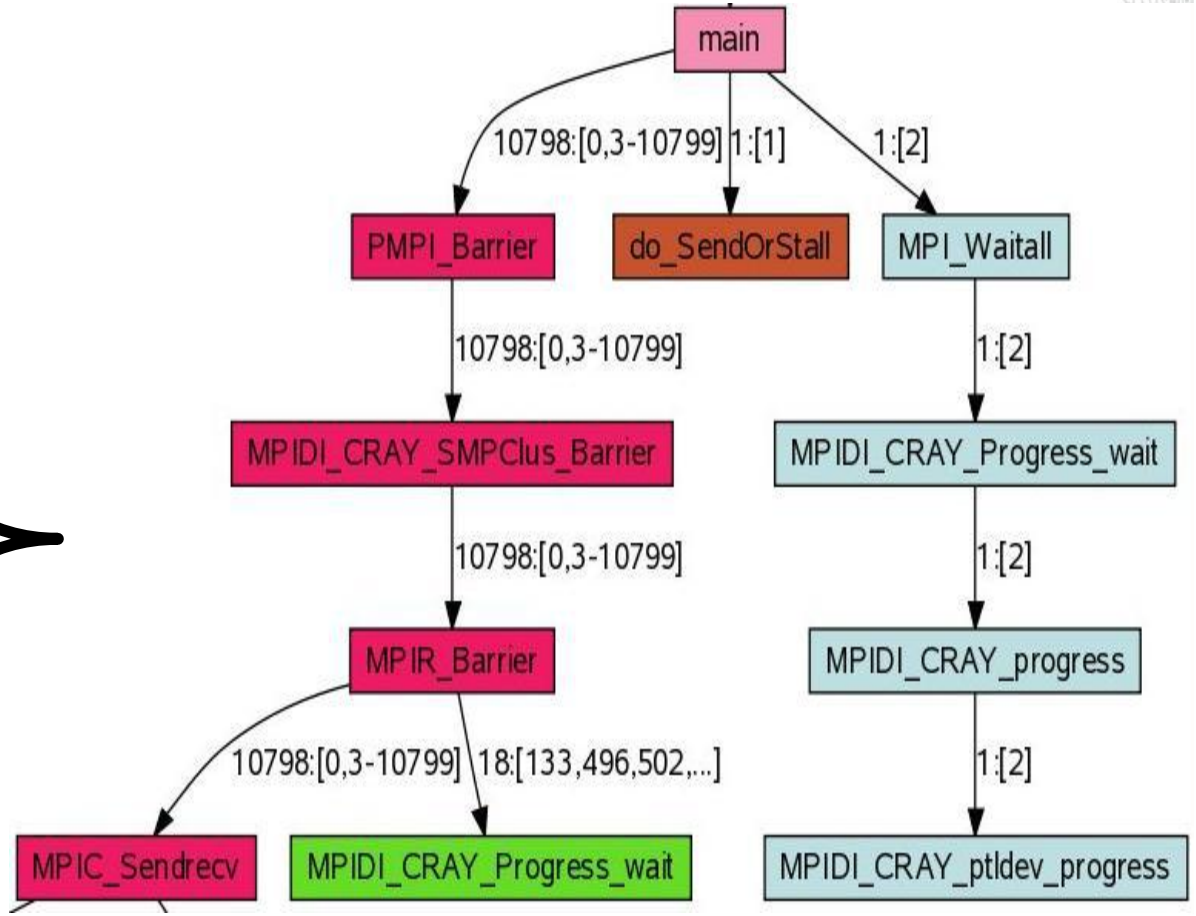


Stack Trace Merge Example

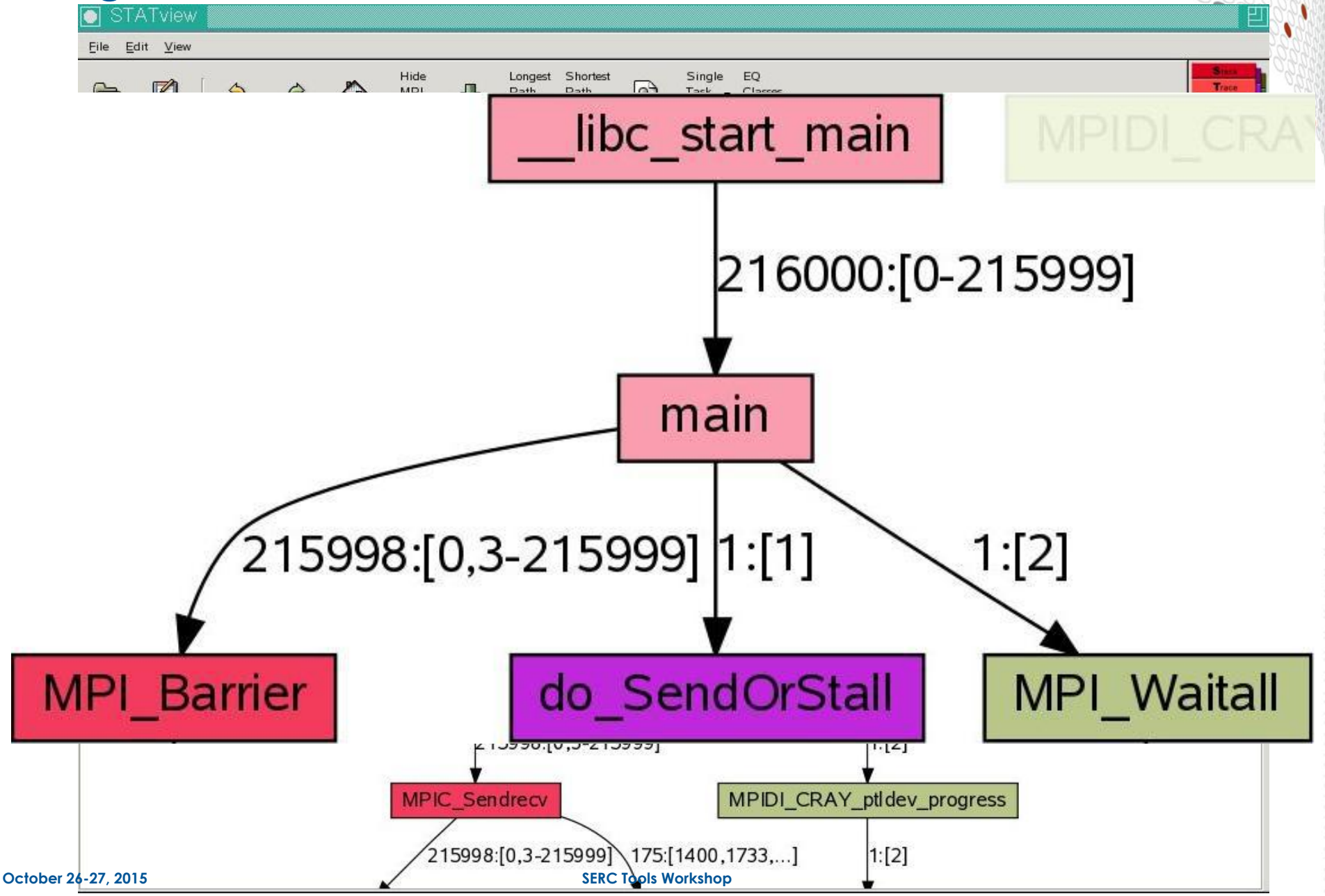


2D-Trace/Space Analysis





Merged Stack



Using STAT to examine a hung job

```
> qsub MYPBS (contains aprun –n ... ./<exe>
> module load stat
```

- Launch your application via a PBS session.
- Load the `stat` module

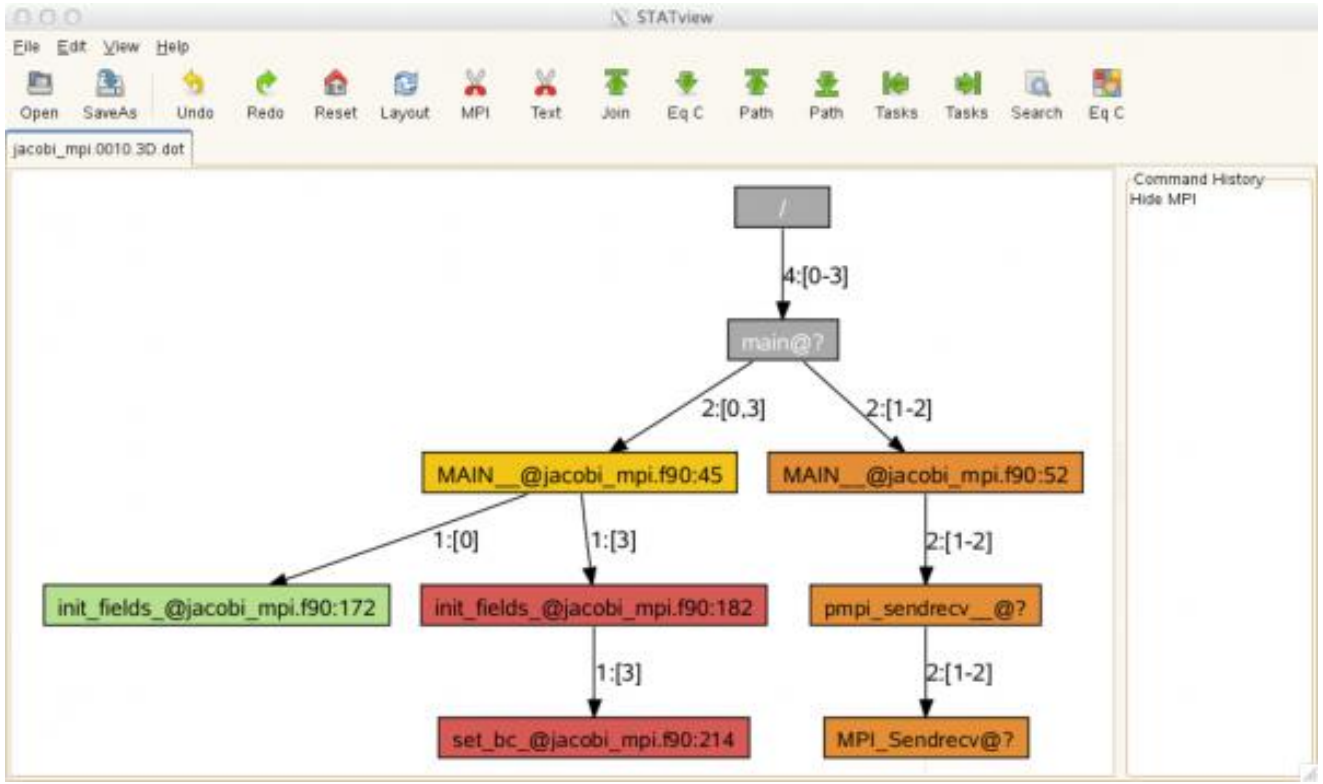
```
>  ps –fu pburgess | grep aprun
pburgess 25861 25852  0 17:57 ?00:00:00 aprun -n 4-d1 -j1 ./jacobi_mpi
pburgess 25880 24422  0 17:58 pts/7    00:00:00 grep aprun
>  STAT -i 25861
Attaching to application...
Attached!
...
Results written to ./stat_results/jacobi_mpi.0010
```

- Wait until application reaches the suspicious state.
- Then launch the command line tool `STAT` with the process id of the `aprun` as an argument and wait until it returns. It writes to the directory "stat_results".
- Terminate the running application with qdel or `kill` the `aprun`

```
> stat-view stat_results/<exe>/<exe>.0000.dot
```

- Now you can start the graphical interface `stat-view` (launches an x-window).
- See next slide for example of output

Looking at STAT output for a hung job



- Rank 0 is in the 'init_fields' routine (line 172 of jacobi_mpi.f90)
- Rank 3 in the 'set_bc' routine (line 214 of jacobi_mpi.f90)
- The other ranks (1 and 2) are in the MPI_Sendrecv function.
- If this pattern persists, the code hangs in these locations. Can now use DDT (or print statements!) to find out why code is stuck here

Abnormal Termination Processing (ATP)

For when things break unexpectedly…
(Collecting back-trace information)

ATP Description

- **Abnormal Termination Processing is a lightweight monitoring framework that detects crashes and provides more analysis instead of silently terminating.**
  - Designed to be so light weight it can be used all the time with almost no impact on performance.
  - Almost completely transparent to the user
    - Requires `atp module` loaded during compilation (usually included by default)
    - Output controlled by the `ATP_ENABLED` environment variable (set by user).
  - Tested at scale (tens of thousands of processors)
- **ATP rationalizes parallel debug information into three easier to use forms:**
  1. A single stack trace of the first failing process to stderr
  2. A visualization of every processes stack trace when it crashed
  3. A selection of representative core files for analysis

ATP Usage

```
export ATP_ENABLED=1
ulimit -c unlimited
```

- Job scripts must include the changes above. Note that ATP respects ulimits on corefiles.
- After abnormal termination the application will not simply crash but proceed with the ATP analysis instead.
- Backtrace of first crashing process is passed to stderr and the merged backtrace of all procs is in `atpMergedBT.dot`

```
Application 867282 is crashing. ATP analysis proceeding...

Stack walkback for Rank 16 starting:
  [empty]@0xffffffffffffffff
  funcA@crash.c:8
Stack walkback for Rank 16 done
Process died with signal 11: 'Segmentation fault'
Forcing core dumps of ranks 16, 0
View application merged backtrace tree with: statview atpMergedBT.dot
You may need to: module load stat

_pmiu_daemon(SIGCHLD): [NID 00752] [c3-0c2s12n0] [Tue Feb 12 19:08:18 2013]
 PE RANK 0 exit signal Segmentation fault
[NID 00752] 2013-02-12 19:08:18 Apid 867282: initiated application terminat
ion
_pmiu_daemon(SIGCHLD): [NID 00753] [c3-0c2s12n1] [Tue Feb 12 19:08:18 2013]
 PE RANK 16 exit signal Segmentation fault
Application 867282 exit codes: 139
Application 867282 resources: utime ~2s, stime ~2s
slurm-10340.out lines 1-16/16 (END)
```

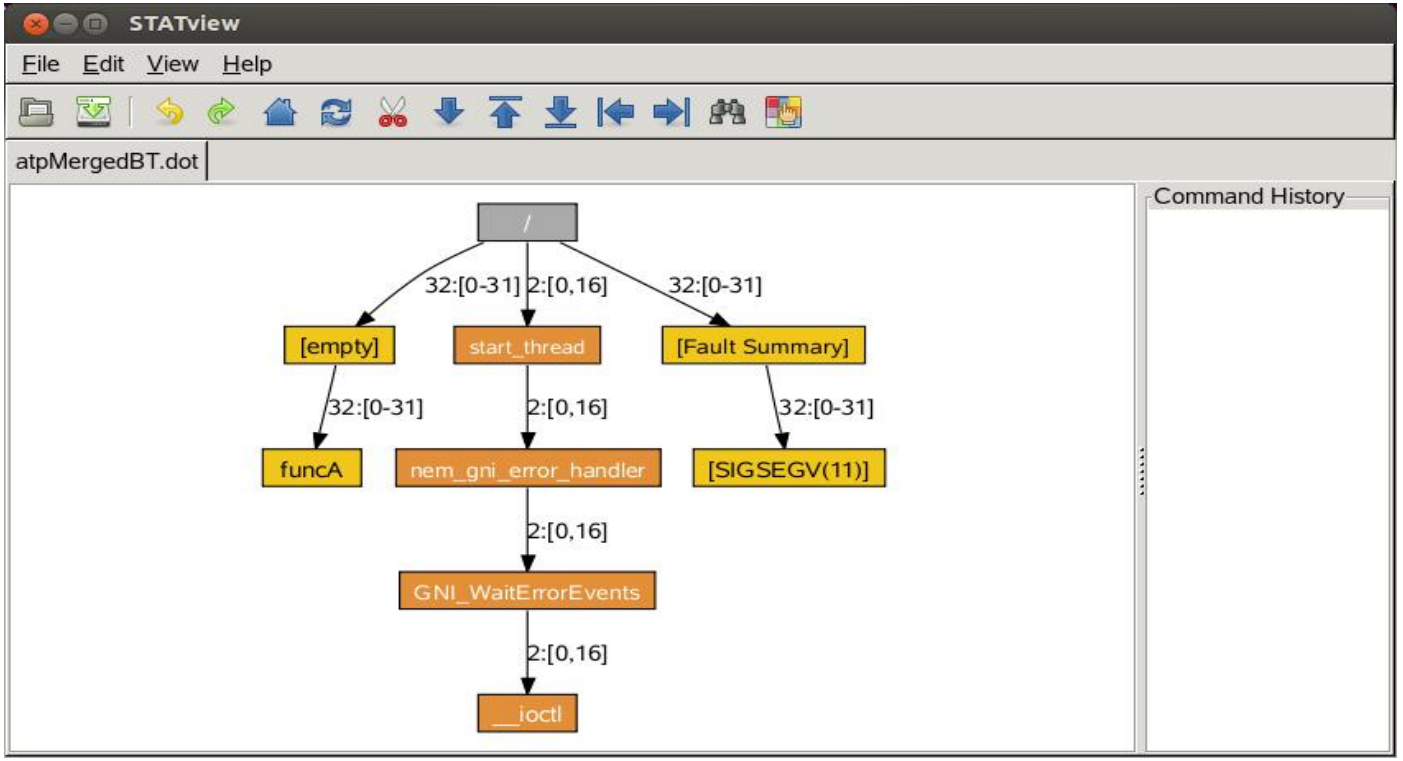
Trace back of crashing process

Core files are being generated.

Viewing the results after the crash

```
> module load stat
> stat-view atpMergedBT.dot
```

- The merged backtrace is inspected via STAT.



- The core files can be inspected with gdb or `Allinea DDT`.

Few Additional ATP Pointers

- **If using Fortran and Intel compiler:**
  - setenv FOR_IGNORE_EXCEPTIONS true # for csh/tcsh
  - export FOR_IGNORE_EXCEPTIONS=true # for bash/sh/ksh
- **If using Fortran and GNU compiler:**
  - Link with –fno-backtrace option
- **man intro_atp**
- **Can also use to debug hung application**

```
export ATP_ENABLED=1
% apstat              # find the 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
```

LGDB

Diving in through the command line…

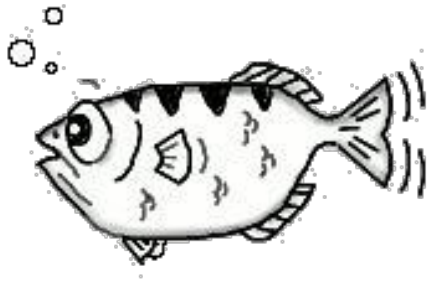
lgdb - Command line debugging

- **LGDB is a line mode parallel debugger for Cray systems available through the module `cray-lgdb`.**
  - It has many of the features of the standard GDB debugger, but includes extensions for handling parallel processes.
  - Binaries should be compiled with debugging enabled, e.g., –g
    - Or for Fast-Track Debugging see later
  - The 2.0 update has introduced new features like comparative debugging. An extensive tutorial is given in http://docs.cray.com/books/S-0042-22/
  - CCDB extends the comparative debugging capabilities of lgdb with a Graphical User Interface.
  - Meant to complement, not replace, DDT – not as feature-rich as commercial debuggers!
  - Under active development!

lgdb - Overview

- **Traditional parallel debugger**
  - Compilers: CCE, PGI, GNU, Intel
  - Languages: C, C++, Fortran
  - Programming models: MPI, SHMEM, OpenACC
  - Partial support: UPC, OpenMP, CUDA, threading
  - Also supports comparative debugging

- **Built on top of gdb**
  - Modified to support Fortran
  - Subset of gdb-like commands



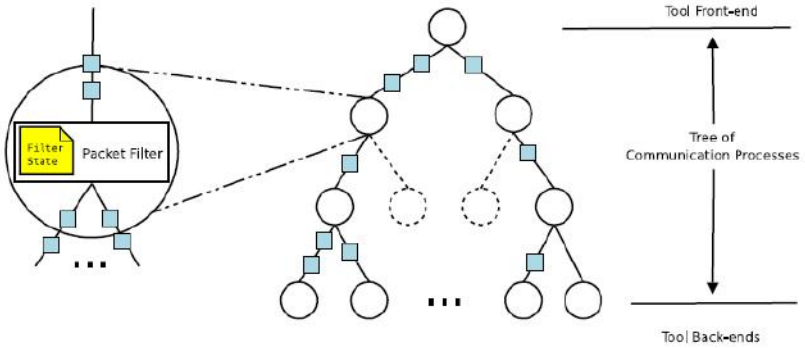
- **Support for attach and launch**
  - Attach – **`attach`** to an existing application
  - Launch – **`launch`** a new application via aprun

- **OpenACC support**
  - pass "-gpu" to `attach/launch` commands

lgdb - Overview (cont.)

- **Utilizes process sets**
  - Define subsets of ranks
  - Restrict focus to operate only on subset
  - Aggregates results based on process sets
  - Correlates to actual PE/rank number (post reordering)

- **Scalable**
  - Utilizes MRNet for its communication network
  - Tested to ~300,000 ranks
  - Good performance at scale, sub-second in many cases



lgdb - Getting Started

- module load cray-lgdb
- man lgdb
- lgdb

```
[andrewg@kaibab] ~ $ module load cray-lgdb
[andrewg@kaibab] ~ $ lgdb
lgdb 2.2 - Cray Line Mode Parallel Debugger
With Cray Comparative Debugging Technology.
Copyright 2007-2013 Cray Inc. All Rights Reserved.
Copyright 1996-2013 Monash University. All Rights Reserved.

Type "help" for a list of commands.
dbg all>
```

- help is your friend

lgdb - Getting Started (cont.)

- **Key how-to concepts of lgdb**
  - Launching application
  - Attaching to application
  - Basic commands
  - Kill/release applications
  - Quitting debugger

lgdb - Launching

- **Step 1: Modify existing PBS job script**
- **Step 2: Determine process set handle**
  - Choose a name (ex. $a)
  - Add in number of PEs (ex. $a{32}) for `launch`
    - This is passed directly to aprun via –n
- **Step 3: Determine additional arguments**
  - See `help launch` for more information

lgdb – Launching (cont.)

Simple example: Modify job script

```
#!/bin/sh

#PBS -V
#PBS -j oe
#PBS -l walltime=1:00:00
#PBS -l select=2

cd $PBS_O_WORKDIR

#cray_debug_start      ← Add this
aprun -n32 a.out
#cray_debug_end        ← Add this
```

lgdb – Launching (cont.)

Simple example: PBS launch

```
dbg all> launch $a{32} --qsub=sample.pbs a.out
Submitting job to the batch system.
Waiting for application to start, please wait...
Starting alps application, please wait...
Creating MRNet communication network...
Waiting for debug servers to attach to MRNet communications network...
Timeout in 60 seconds. Please wait for the attach to complete.
Number of dbgsrvs connected: [1];  Timeout Counter: [0]
Number of dbgsrvs connected: [1];  Timeout Counter: [1]
Number of dbgsrvs connected: [32];  Timeout Counter: [0]
Finalizing setup...
Launch complete.
a{0..31}: Initial breakpoint, main at /lus/nid00030/andrewg/
test_fixtures/c_type_test.c:131
```

lgdb – Launching (cont.)

Simple example: PBS launch with `session`

```
dbg all> session --qsub=sample.pbs
Submitting job to the batch system.
Session ready.
dbg all> launch $a{32} a.out
Waiting for application to start, please wait...
…
Launch complete.
a{0..31}: Initial breakpoint, main at /lus/nid00030/andrewg/
test_fixtures/c_type_test.c:131
dbg all> launch $b{32} a.out
Waiting for application to start, please wait...
…
Launch complete.
b{0..31}: Initial breakpoint, main at /lus/nid00030/andrewg/
test_fixtures/c_type_test.c:131
```

lgdb – Launching (cont.)

`session` versus `--qsub` in `launch`

- Use **`session`** when:
  - Single PBS job reservation, multiple **`launch`**
  - Multiple kill/launch commands
- Use **`--qsub`** in **`launch`** when:
  - Very specific/complex job script
  - Don't mind having reservation go away

lgdb - Attaching

- **Step 1: Determine Apid**
  - Use **`apstat`** utility
  - This is not the pid of the aprun process
- **Step 2: Determine process set handle**
  - Choose a name (ex. $a)
  - Do not add in number of PEs for **`attach`**
    - Automatically determined from ALPS info
- **Step 3: Determine additional arguments**
  - See **`help attach`** for more information
  - Only argument currently available is **`--gpu`**

lgdb – Attaching (cont.)

- **Simple example: Determine Apid from apstat**

```
[andrewg@kaibab] test_fixtures $ apstat
Compute node summary
    arch config     up   resv    use  avail   down
      XT     84     84      8      8     76      0

No pending applications are present

Total placed applications: 6
    Apid ResId     User PEs Nodes    Age State   Command
12572058   161   n13942   1     1 17h02m   run    hybrid
12572167   162   n13942   1     1 16h53m   run    hybrid
12573589   711   n13942   1     1 12h43m   run    hybrid
12573723   779   n13942   1     1 12h29m   run    hybrid
12582705  1181 jdykstra   2     2  0h32m   run ccmlaunch
12582710  1185  andrewg  32     2  0h00m   run     a.out ← Copy this Apid
```

lgdb – Attaching (cont.)

- **Simple example: Attaching**

```
dbg all> attach $a 12582710
Attaching to application, please wait...
Creating MRNet communication network...
Waiting for debug servers to attach to MRNet communications network...
Timeout in 60 seconds. Please wait for the attach to complete.
Number of dbgsrvs connected: [1];  Timeout Counter: [0]
Number of dbgsrvs connected: [1];  Timeout Counter: [1]
Number of dbgsrvs connected: [32];  Timeout Counter: [0]
Finalizing setup...
Attach complete.
Current rank location:
a{0,24}: #0  0x000000000055d7d1 in nanosleep at /usr/src/packages/BUILD/
glibc-2.11.3/nptl/../sysdeps/unix/syscall-template.S:82
a{0,24}: #1  0x000000000058ccec in __sleep at /usr/src/packages/BUILD/
glibc-2.11.3/posix/../nptl/sysdeps/unix/sysv/linux/sleep.c:138
a{0,24}: #2  0x0000000000401913 in main at /lus/nid00030/andrewg/
test_fixtures/c_type_test.c:226
a{1..23,25..31}: #0  0x000000000055d7b0 in __nanosleep_nocancel at /usr/src/
packages/BUILD/glibc-2.11.3/nptl/../sysdeps/unix/syscall-template.S:82
a{1..23,25..31}: #1  0x000000000058ccec in __sleep at /usr/src/packages/
BUILD/glibc-2.11.3/posix/../nptl/sysdeps/unix/sysv/linux/sleep.c:138
a{1..23,25..31}: #2  0x0000000000401913 in main at /lus/nid00030/andrewg/
test_fixtures/c_type_test.c:226
```

lgdb – Attaching (cont.)

- **Simple example: Attaching (cont.)**

```
dbg all> viewset
Name        Procs
all         a{0..31}
a           a{0..31}
```

lgdb – Example commands

- **break**
  - Set a breakpoint in every rank

```
dbg all> break c_type_test.c:226
a{0..31}: Breakpoint 1: file /lus/nid00030/andrewg/test_fixtures/c_type_test.c, line
226.
```

- **continue**
  - Continue execution of application
  - Runs until breakpoint encountered or exit()

```
dbg all> continue
<$a>: Hello World! from process 0      ← Interleaved stdout from application $a
…
<$a>: Hello World! from process 23
a{0..31}: Breakpoint 1, main at /lus/nid00030/andrewg/test_fixtures/c_type_test.c:226
<$a>: Hello World! from process 24
…
<$a>: Hello World! from process 31
dbg all>
```

lgdb – Example commands (cont.)

- **info locals**
  - Display all local symbols in scope

```
dbg all> info locals
a{0..31}: Name:argc                    Type:int
a{0..31}: Name:argv                    Type:signed char **
a{0..31}: Name:myRank                  Type:int
a{0..31}: Name:numProcs                Type:int
…
```

- **print**
  - Print the value of a local variable
  - Aggregates results if possible
  - Optionally can specify procset to reduce information

```
dbg all> print numProcs
a{0..31}: 32
dbg all> print $a::myRank
a{0}: 0
a{1}: 1
a{2}: 2
a{3}: 3
…
a{31}: 31
```

lgdb – Example commands (cont.)

- **print (cont.)**

```
dbg all> defset $test $a{0},$a{2},$a{8..10}
dbg all> print $test::myRank
test{0}: 0
test{2}: 2
test{8}: 8
test{9}: 9
test{10}: 10
```

- **list**
  - Display source code lines at current breakpoint

```
dbg all> list
a{0..31}: 226           sleep(1000);
a{0..31}: 227
a{0..31}: 228           MPI_Finalize();
a{0..31}: 229
a{0..31}: 230           return 0;
a{0..31}: 231   }
a{0..31}: 232
```

LGDB – Process Groups

```
dbg all> bt
all[0,2..31]: #0  0x0000000000400979 in main at
/tdsnfs1/y02/y02/ted/xthi.c:47
all[1]: #0  0x0000000000400984 in main at
/tdsnfs1/y02/y02/ted/xthi.c:48
```

- Output from commands is grouped into common sets, e.g., backtraces (`bt`) will be prepended with groups.

```
defset $<newgrp> $<pset>{rank1},$<pset>{rank37}
```

- Debugging commands are issued in parallel to all processes in the "focus" group. By default this is `$<pset>`, i.e. all the processors in the application.
- New groups can be created with `defset`.

```
focus $<newgrp>
```

- The focus can be changed with the `focus` command.

lgdb – Kill/Release

- **Killing application**
  - Must have been launched/attached
  - Will send an **`apkill -9`** to the application

```
dbg all> kill $a
Shutting down debugger and killing application for 'a'.
dbg all>
```

- **Releasing application**
  - Must have been launched/attached
  - Will detach the debugger from application
  - Continues execution

```
dbg all> release $a
Shutting down debugger and resuming application for 'a'.
dbg all>
```

lgdb - Exiting

- To exit, use `quit` command
  - Any launched applications will be killed
  - Any attached applications will be released

```
dbg all> quit
Shutting down debugger and killing application for 'a'.
```

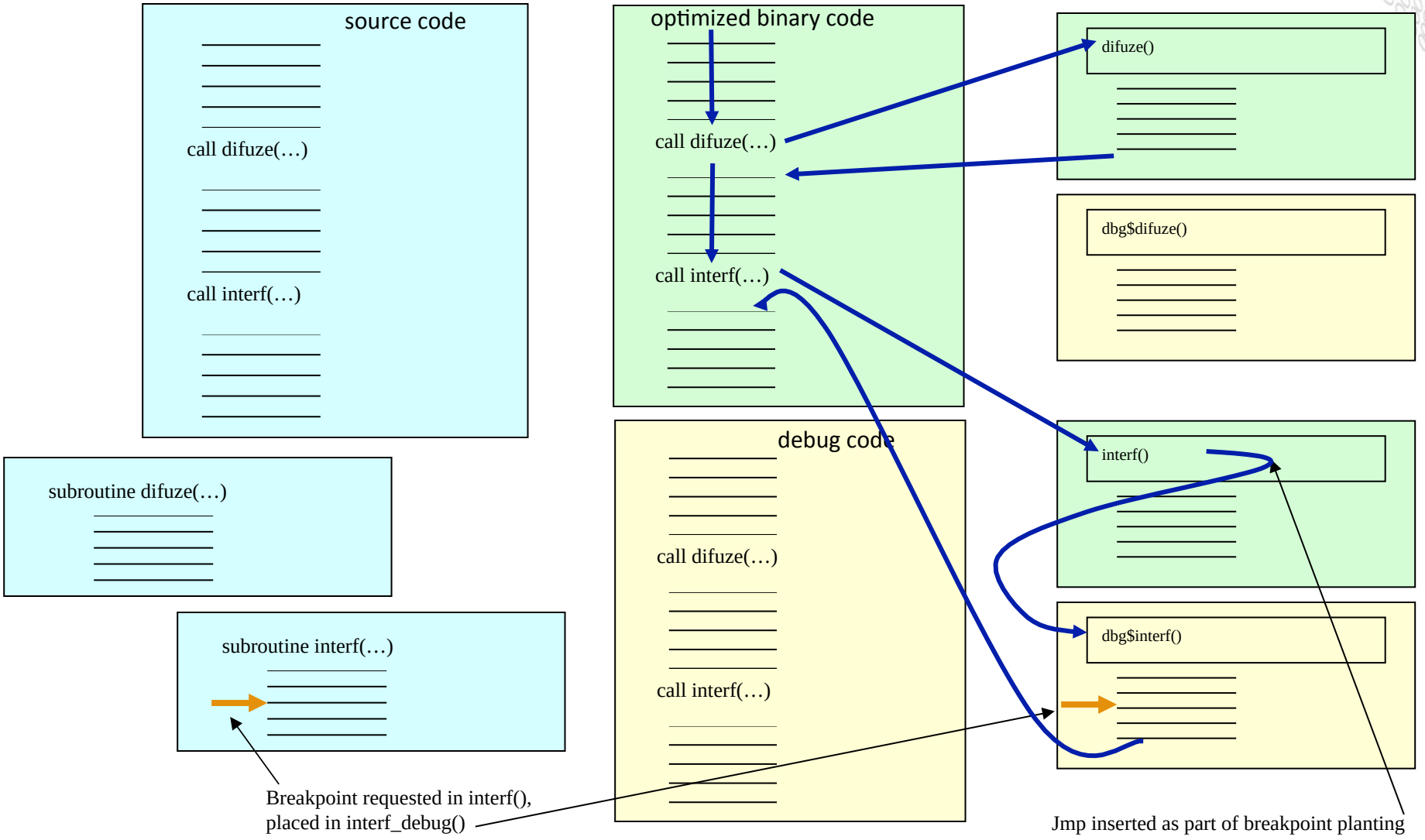
Fast Track Debugging

For getting to the problem more quickly…

The Problem ... And a Solution

- **Compiling with debug support eliminates optimizations**
  - Today's machines require high levels of optimization for efficient execution.
  - Debugging notably slows down the execution.
  - Problem might disappear with lower optimization.
- **Idea behind fast track debugging**
  - Compile such that both debug and non-debug (optimized) versions of each routine are created. Use `-Gfast` instead of `-g` with the Cray compiler for that purpose. Check the man pages of Cray compilers.
  - Linkage such that optimized versions are used by default – so can run at optimized-code speed but with full DWARF information
  - Debugger overrides default linkage when setting breakpoints and stepping into functions
  - Supported by Allinea DDT and lgdb.

A Closer Look at How FTD Works



Comparative Debugging

What on earth just went wrong?
It was running yesterday....
It ran with the Intel Compiler, why not CCE?
It ran before I made that innocent little change...
Surely that library change can't have hurt things?

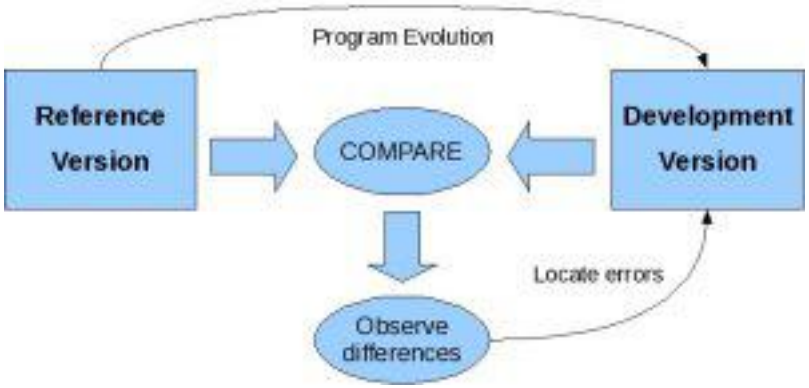
ccdb – VERY Brief Overview

- **Comparative debugging tool**
  - NOT a traditional debugger!
  - GUI written in perl
  - Assists with comparative debugging

- **Leverages lgdb**
  - lgdb is sitting underneath
  - Problem: comparative debugging with lgdb is hard
    - Command line based
    - Formal language
    - Potentially hundreds of lines of commands
  - Hides the complexity behind GUI elements

ccdb – Overview

- **What is comparative debugging?**
  - Data centric approach
  - Two applications, same data
  - Key idea: The data should match
  - Doesn't actually locate bugs
  - Quickly isolate deviating variables



- **How does this help me?**
  - Algorithm re-writes
  - Language ports
  - Different libraries/compilers
  - New architectures

Summary

- **The Cray XC40 provides a series of light weight debugging tools which can be efficiently used for large simulations and different scenarios.**
  - Stack Trace Analysis Tool (**STAT**)
  - Abnormal Termination Processing **(ATP**)
  - Fast Track Debugger (**FTD**)

- **These tools provide only a rough view of the problem.**
  - This might be sufficient for a developer who is very familiar with the code.
  - If a code is very large and a user does not have sufficient experience, a more sophisticated tool like Allinea DDT can help.
  - Core files out of ATP and binaries compiled with FTD can be inspected and debugged with these more advanced tools.