STAT: the Stack Trace Analysis Tool

Gregory L. Lee

Dorian C. Arnold

Dong H. Ahn

Bronis R. de Supinski

Barton P. Miller

Martin Schulz

STAT: the Stack Trace Analysis Tool

by Gregory L. Lee

by Dorian C. Arnold

by Dong H. Ahn

by Bronis R. de Supinski

by Barton P. Miller

by Martin Schulz

Table of Contents

Disclaimer

Auspice

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

License

Produced at the Lawrence Livermore National Laboratory

Written by Gregory Lee [lee218@llnl.gov], Dorian Arnold, Dong Ahn, Bronis de Supinski, Barton Miller, and Martin Schulz.

LLNL-CODE-400455.

This file is part of STAT. For details, see http://www.paradyn.org/STAT. Please also read STAT/LICENSE.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer (as noted below) in the documentation and/or other materials provided with the distribution.

Neither the name of the LLNS/LLNL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LAWRENCE LIVERMORE NATIONAL SECURITY, LLC, THE U.S. DEPARTMENT OF ENERGY OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Additional BSD Notice

1. This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 with the DOE.

2. Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.

3. Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Chapter 1. Introduction

The Stack Trace Analysis Tool (STAT) is a highly scalable, lightweight debugger for parallel applications. STAT was initially developed as a collaboration between the Lawrence Livermore National Laboratory and the University of Wisconsin. It is currently open source software released under the Berkeley Software Distribution (BSD) license. It builds on a highly portable, open source infrastructure, including LaunchMON for tool daemon launching, MRNet for scalable communication, and StackWalker for obtaining stack traces.

STAT works by gathering stack traces from all of a parallel application's processes and merging them into a compact and intuitive form. The resulting output indicates the location in the code that each application process is executing, which can help narrow down a bug. Furthermore, the merging process naturally groups processes that exhibit similar behavior into process equivalence classes. A single representative of each equivalence can then be examined with a full-featured debugger like TotalView[1] or DDT[2] for more in-depth analysis.

STAT has been ported to several platforms, including Linux clusters, IBM's Bluegene/L and Bluegene/P machines, and Cray XT systems. It works for Message Passing Interface (MPI) applications written in C, C++, and Fortran and also supports threads. STAT has already demonstrated scalability over 200,000 MPI tasks and its logarithmic scaling characteristics position it well for even larger systems.

Notes

1. http://www.totalviewtech.com/
2. http://www.allinea.com/index.php?page=48

Chapter 2. Overview

STAT, the Stack Trace Analysis Tool, helps isolate bugs by gathering stack traces from each individual process of a parallel application and merging them into a global, yet compact representation. Each stack trace, as depicted in Figure 2-1, captures the function calling sequence of an individual process. The nodes are labeled with the function names and the directed edges show the function calling sequence from caller to callee. STAT's stack trace merging process forms a call graph prefix tree, which can be seen in Figure 2-1. The prefix tree groups together traces from different processes that have the same calling sequence and labels the edges with the count and set of tasks that exhibited that calling sequence. Nodes in the prefix tree that are visited by the same set of tasks are given the same color, providing the user with a quick means of identifying the various process equivalence classes.

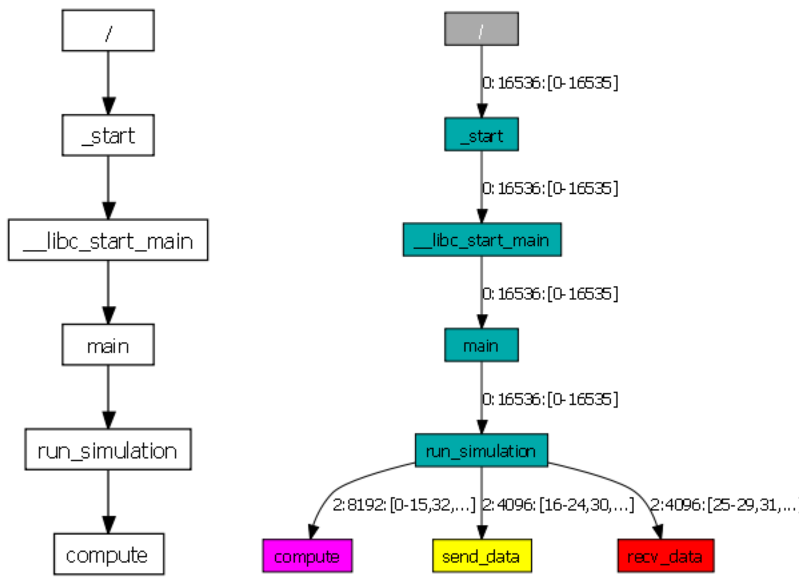

Figure 2-1. A single stack trace (left) and a STAT merged call prefix tree (right)

STAT merges stack traces into 2D spatial and 3D spatial-temporal call prefix trees. The 2D spatial call prefix tree (Figure 2-2) represents a single snapshot of the entire application. The 3D spatial-temporal call prefix tree (Figure 2-3) takes a series of snapshots from the application over time and is useful for analyzing time-varying behavior.

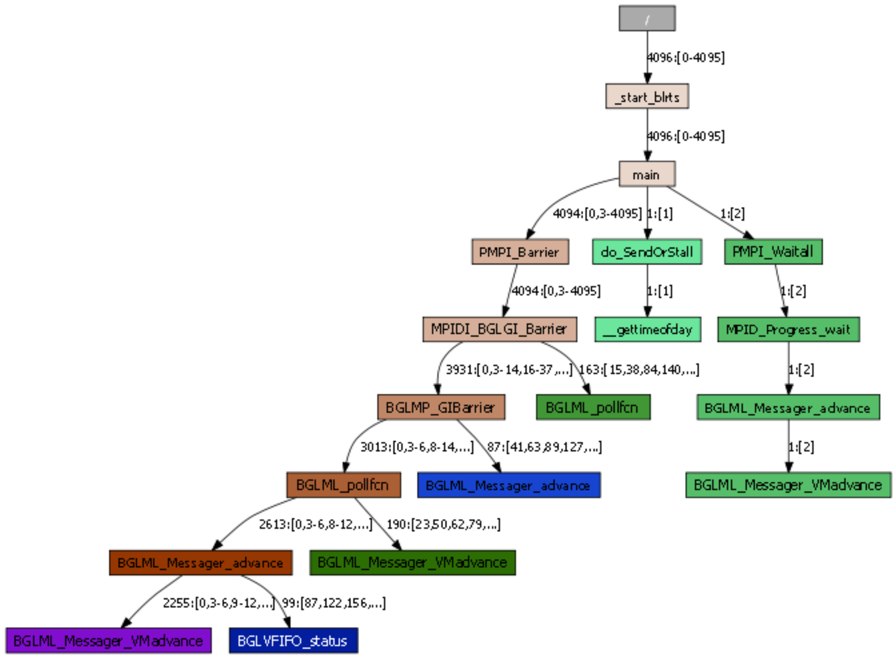

Figure 2-2. A 2D spatial call prefix tree

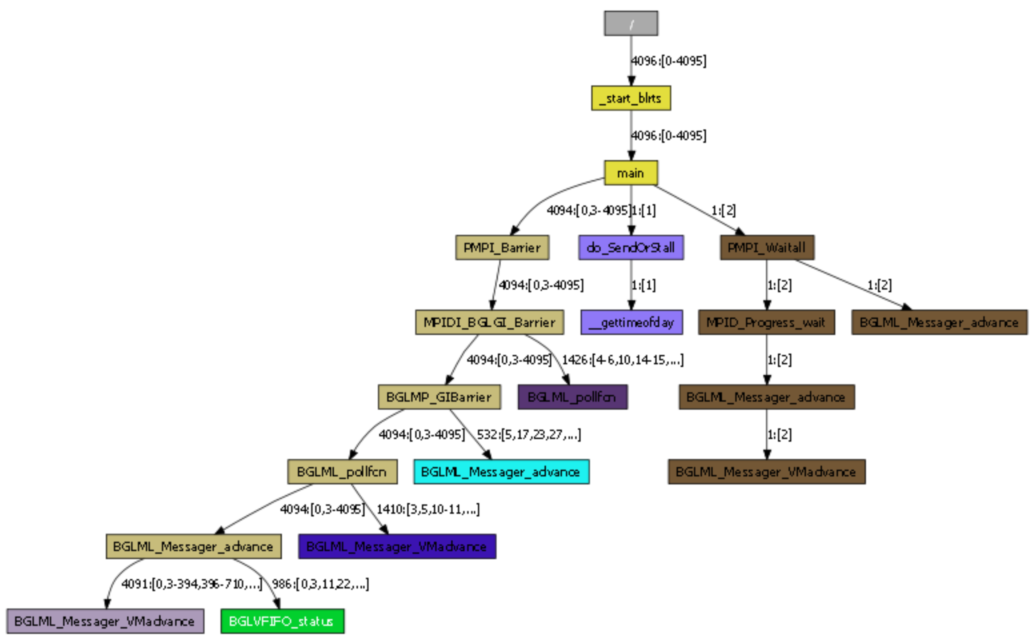

Figure 2-3. A 3D spatial call prefix tree

Stack traces based on function names only provide a high-level overview of the application's execution. However, for certain bugs this view may be too coarse-grained so STAT is also capable of gathering stack traces with more fine-grained information. In particular, STAT can also record the program counter of each frame or with the appropriate debug information compiled into the application (i.e., with the "-g" compiler flag), STAT can gather the source file and line number of each stack frame. Both of these refinements can further delineate processes and refine the process equivalence classes.

In addition, line number information can be fed into a static code analysis engine to derive the logical temporal order of the MPI tasks Figure 2-4. This analysis traverses from the root of the tree towards the leaves, at each step analyzing the control flow of the source code and sorting sibling nodes by the amount of execution progress made through the code. For straight-line code, this simply means that one task has made more progress if it has executed past the point of another task, i.e., if it has a greater line number. This ordering is partial since two tasks in different branches of an if-else are incomparable. In cases where the program points being compared are within a loop, STAT can extract the loop ordering variable from the application processes and further delineate tasks by execution progress. This analysis is useful for identifying the culprit in a deadlocked or livelocked application, where the problematic task has often either made the least or most progress through the code, leaving the remaining tasks stuck in a barrier or blocked pending a message. Note, this feature is still a prototype. Please contact Greg Lee for an experimental version.

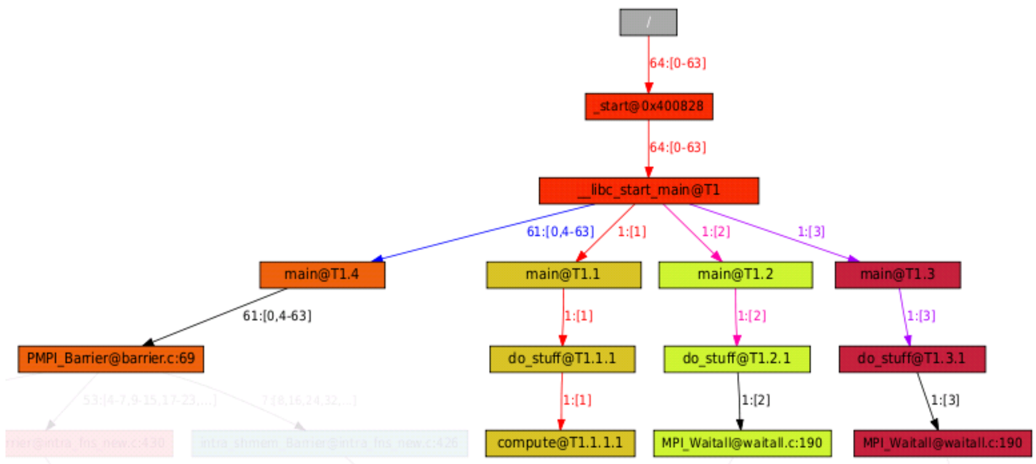

Figure 2-4. STAT's temporal ordering analysis engine indicates that task 1 has made the least progress. In this example, task 1 is stuck in a compute cycle, while the other tasks are blocked in MPI communication, waiting for task 1.

Chapter 3. Installing STAT

Dependent Packages

STAT has several dependencies

Table 3-1. STAT Dependent Packages

Package Package Web Page	What It Does
Graphlib	Graph creation, merging, and export
https://outreach.scidac.gov/projects/stat/	
Launchmon	Scalable daemon co-location
http://sourceforge.net/projects/launchmon/	
Libdwarf	Debug information parsing (Required by StackWalker)
http://reality.sgiweb.org/davea/dwarf.html	
MRNet	Scalable multicast and reduction network
http://www.paradyn.org/mrnet/	
StackWalker	Lightweight stack trace sampling
http://www.paradyn.org/html/downloads.html	

In addition, the STAT GUI requires Python[1] with PyGTK[2], both of which are commonly preinstalled with many Linux operating systems. The Pygments[3] Python module can optionally be installed to allow the STAT GUI to perform syntax highlighting of source code.

Installation

First run configure. You will need to use the --with-package options to specify the install prefix for mrnet, graphlib, launchmon, libdwarf, and stackwalker. These options will add the necessary includes and library search paths to the compile options. Refer to configure --help for exact options. You may also wish to specify the maximum number of communication processes to launch per node with the option --with-procspernode=number, generally set to the number of cores per node.

STAT will use StackWalker by default. However, it can use Dyninst instead if you specify --with-dyninst to the configure script. Further, STAT will also use MRNet v2.1 by default. You may use MRNet v1.X by specifying --with-mrnet1 and MRNet v2.2 by specifying --with-mrnet22.

To enable the building of the STAT GUI, configure with --enable-gui and also specify --with-python-include-dir=dir, where dir contains Python.h.

On BlueGene systems, also be sure to configure --with-bluegene. This will enable the BGL macro for BlueGene specific compilation.

Next you just need to run:

```
make
make install
```

Note that STAT hardcodes the paths to its daemon and filter shared object, assuming that they are in $prefix/bin and $prefix/lib respectively, thus testing should be done in the install prefix after running "make install" and the installation directory should not be moved. The path to these components can, however, be overridden with the --daemon and --filter arguments. Refer to STAT --help or the STAT man page for more details.

Notes

1. http://www.python.org/
2. http://www.pygtk.org/
3. http://pygments.org/

Chapter 4. Using the STAT Command

Description

STAT (the Stack Trace Analysis Tool) is a highly scalable, lightweight tool that gathers and merges stack traces from all of the processes of a parallel application. After running the STAT command, STAT will create a STAT_results directory in your current working directory. This directory will contain a subdirectory, based on your parallel application's executable name, with the merged stack traces in DOT format.

STAT Options

-a, --autotopo

let STAT automatically create topology.

-f, --fanout *`width`*

Sets the maximum tree topology fanout to *`width`*. Specify nodes to launch communications processes on with `--nodes`.

-d, --depth *`depth`*

Sets the tree topology depth to *`depth`*. This option takes precedence over the `--fanout` option. Specify nodes to launch communications processes on with `--nodes`.

-u, --usertopology *`topology`*

Specify the number of communication nodes per layer in the tree topology, separated by dashes, with *`topology`*. This option takes precedence over the `--fanout` and `--depth` options. Specify nodes to launch communications processes on with `--nodes`. Example topologies: 4, 4-16, 5-20-75.

-n, --nodes *`nodelist`*

Use the specified nodes in *`nodelist`*. To be used with `--fanout`, `--depth`, or `--usertopology`. Example nodes lists: host1; host1,host2; host[1,5-7,9].

-p, --procs *`processes`*

Sets the maximum number of communication processes to be spawned per node to *`processes`*. This should typically be set to the number of CPUs per node.

-j, --jobid *`id`*

Append *`id`* to the output directoory and file prefixes. This is useful for associating STAT results with a batch job.

-r, --retries *`count`*

Attempt *`count`* retries per sample to try to get a complete stack trace.

-R, --retryfreq *`frequency`*

Wait *`frequency`* milliseconds between sample retries. To be used with the `--retries` option.

-P, --withpc

Sample program counter values in addition to function names.

-i, --withline

Sample source line number in addition to function names.

-c, --comprehensive

Gather 4 traces: function only; function + line; function + PC; and 3D function only.

-w, --withthreads

Sample helper threads in addition to the main thread.

-t, --traces `count`

Gather `count` traces per process.

-T, --tracefreq `frequency`

Wait `frequency` milliseconds between samples. To be used with the `--traces` option.

-S, --sampleindividual

Save all individual samples in addition to the 3D trace when using `--traces` option.

-C, --create `arg_list`

Launch the application under STAT's control. All arguments after -C are used to launch the app. Namely, `arg_list` is the command that you would normally use to launch your application.

-D, --daemon `path`

Specify the full path `path` to the STAT daemon executable. Use this only if you wish to override the default.

-F, --filter `path`

Specify the full path `path` to the STAT filter shared object. Use this only if you wish to override the default.

-s, --sleep `time`

Sleep for `time` seconds before attaching and gathering traces. This gives the application time to get to a hung state.

-l, --log

[`FE` | `BE` | `ALL`]

Enable debug logging of the `FE`, `BE`, or `ALL`.

-L, --logdir `log_directory`

Dump logging output into `log_directory`. To be used with the `--log` option.

STAT Usage Example

The most typical usage is to invoke STAT on the job launcher's PID:

```
% srun mpi_application arg1 arg2 &
[1] 16482
```

```
% ps
  PID TTY          TIME CMD
16755 pts/0    00:00:00 bash
16842 pts/0    00:00:00 srun
16871 pts/0    00:00:00 ps

% STAT 16482
```

You can also launch your application under STAT's control with the -C option. All arguments after -C are used for job launch:

```
% STAT -C srun mpi_application arg1 arg2
```

At larger scales, you may want to employ a more scalable tree topology. For example, if you're running on 1024 nodes, you may want to try a fanout of sqrt(1024) = 32. You will need to specify a list of nodes that contains enough processors to accommodate the ceil(1024/32) = 32 communication processes being launched. Be sure that you have login permissions to the specified nodes and that they contain the mrnet_commnode executable and the STAT_FilterDefinitions.so library. STAT may be configured to automatically deploy a tree based on the job size, in which case you do not need to specify a topology.

```
% STAT --fanout 32 --nodes atlas[1-4] --procs 8 16482
```

Upon successful completion, STAT will write its output to a STAT_results directory within the current working directory. Each run creates a subdirectory named after the application with a unique integer ID. STAT's output indicates the directory created with a message such as:

```
Results written to /home/user/bin/STAT_results/mpi_application.6
```

Within that directory will be one or more files with a .dot extension. These .dot files can be viewed with **STATview**.

Chapter 5. Using the STATview GUI

Description

STATview (Figure 5-1) 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. Each node in the STAT call prefix tree represents a function call and the directed edges denote the calling sequence. Further, the edges are labeled by the set of tasks that have taken that call path. For simplification, STATview will display the number of tasks in the set and truncate long task lists in the main display with "..." notation. Similarly, long function names will be truncated with "..." notation. Nodes are colored based on the set of tasks of the incoming edge, providing a visual distinction when different tasks take different branches.

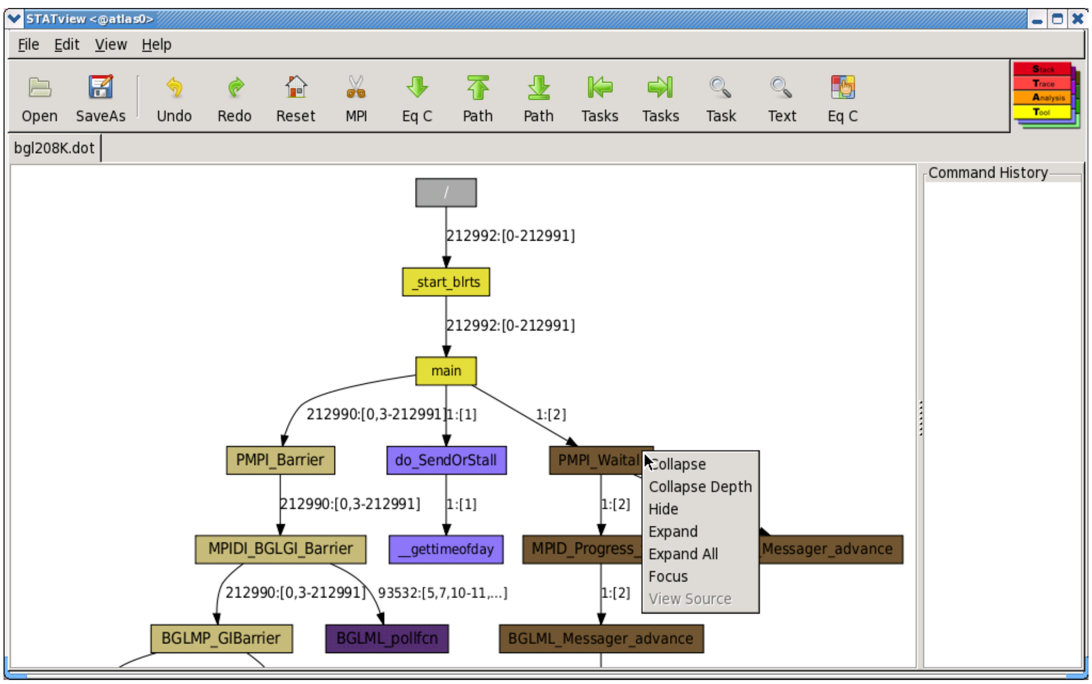

Figure 5-1. A screenshot of the STATview GUI.

The STATview Node Menu

By left clicking on a node in the call prefix tree you will get a window displaying the full list of tasks and the full frame label (Figure 5-2). This window also contains buttons that allow for the manipulation of the graph from that node. Right clicking on a node provides a pop-up menu with the same options. Note all of these operations are performed on the current visible state of the call prefix tree.

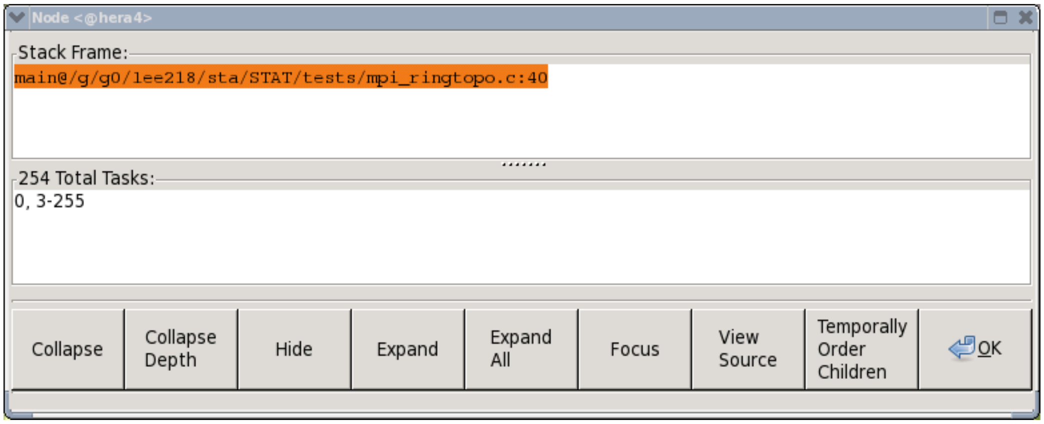

Figure 5-2. The node pop-up window

The node operations are defined as follows:

Collapse

hide all of the descendents of the selected node.

Collapse Depth

collapse the entire tree to the depth of the selected node.

Hide

the same as **Collapse**, but also hides the selected node.

Expand

show (unhide) the immediate children of the selected node.

Expand All

show (unhide) all descendents of the selected node.

Focus

hide all nodes that are neither ancestors nor descendents of the selected node. (Note: This will not unhide any hidden ancestors.)

View Source

creates a popup window (Figure 5-3) displaying the source file (only for stack traces with line number information). This may require the source file's path to be added to the search path, through **File** -> **Add Search Paths**.

Temporally Order Children

(prototype only) determine the temporal order of the node's children (only for stack traces with line number information). Requires the source file's path and all include paths to be added to the search path, through **File** -> **Add Search Paths**.

OK

closes the pop-up window.

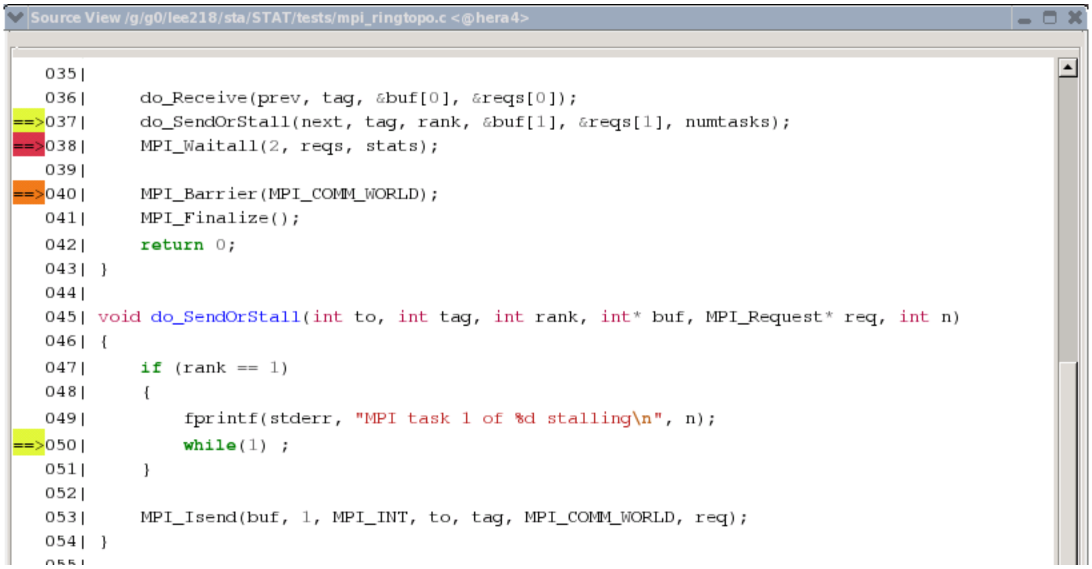

Figure 5-3. The source view window. The colored arrows correspond to the nodes in the call prefix tree.

The STATview Toolbar

The main window also has several tree manipulation options (Figure 5-4). Note the initial click of a traversal operation operates on the original call prefix tree, while the remaining operations are performed on the current visible state of the call prefix tree.

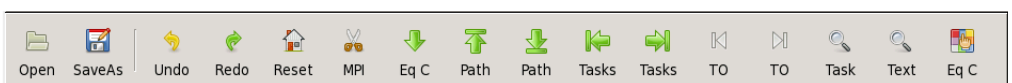

Figure 5-4. The STATview tree manipulation toolbar.

The toolbar operations are defined as follows:

Open

Open a STAT generated .dot file

Save As

Save the current graph in .dot format, which can be displayed by STATview or in an image format, such as PNG or PDF, which can be viewed on any computer with an image viewer

Undo

Undo the previous operation

Redo

Redo the undone operation

Reset

Revert to the original graph

[Cut] MPI

Collapse the MPI implementation frames below the MPI function call

[Traverse] Eq C

Traverse the prefix tree by expanding the leaves to the next equivalence class set. The first click will display the top-level equivalence class.

[Traverse Longest] Path

Traversal focus on the next longest call path(s). The first click will focus on the longest path.

[Traverse Shortest] Path

Traversal focus on the next shortest call path(s). The first click will focus on the shortest path.

[Traverse Least] Tasks

Traversal focus on the path(s) with the next least visiting tasks. The first click will focus on the path with the least visiting tasks.

[Traverse Most] Tasks

Traversal focus on the path(s) with the next most visiting tasks. The first click will focus on the path with the most visiting tasks.

[Traverse Least] TO

Temporal Order traversal focus on the path(s) that have made the least execution progress in the application. The first click will focus on the path that has made the least progress.

[Traverse Most] TO

Temporal Order traversal focus on the path(s) that have made the most execution progress in the application. The first click will focus on the path that has made the most progress.

[Focus] Task

Only display call paths taken by the specified task

[Focus] Text

Only display call paths containing the specified text

[Identify] Eq C

Identify the equivalence classes of the visible graph. After clicking on this button, a window will pop up showing the complete list of equivalence classes.

Chapter 6. Using the STAT GUI

Description

STAT includes a graphical user interface (GUI) to run STAT and to visualize STAT's outputted call prefix trees (Figure 6-1). This GUI provides a variety of operations to help focus on particular call paths and tasks of interest. It can also be used to identify the various equivalence classes and includes an interface to attach a heavyweight debugger to the representative subset of tasks.

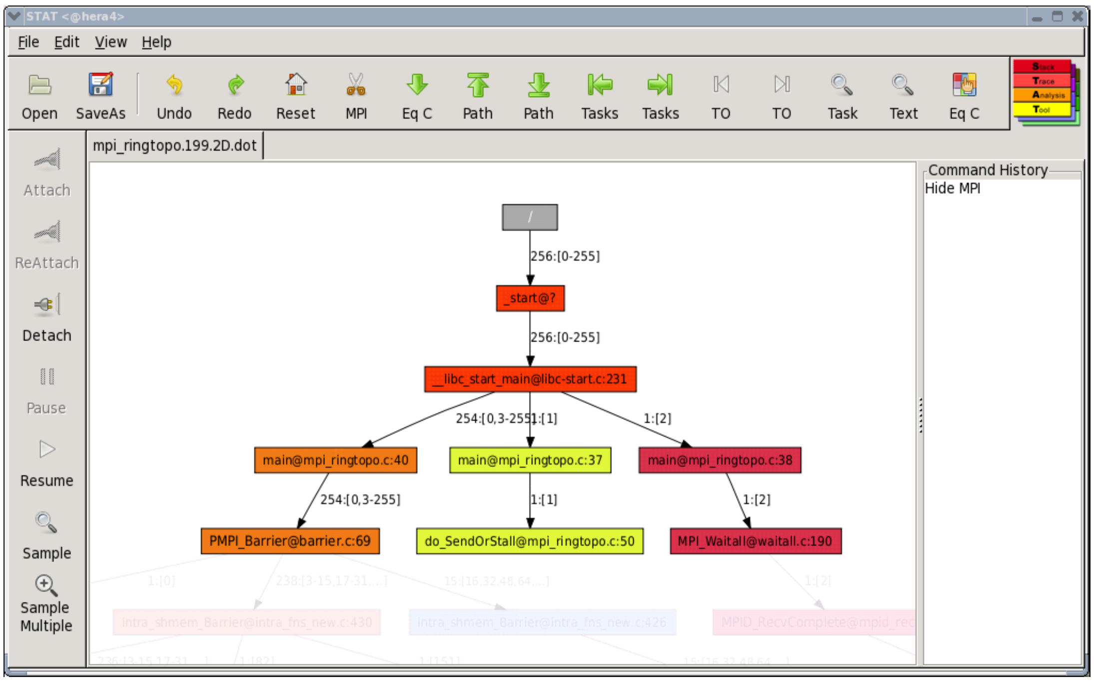

Figure 6-1. A screenshot of the STAT GUI

The STAT GUI Toolbar

In addition to the operations provided by STATview, STATGUI provides a toolbar (Figure 6-2) to control STAT's operation.

Figure 6-2. The STAT GUI toolbar.

Attach

Attach to your parallel application and gather an initial sample.

ReAttach

Reattach to the previous parallel application and gather an initial sample.

Detach

Detach from your parallel application.

Pause

Put the application in a stopped state.

Resume

Set the application running.

Sample

Gather and merge a single stack trace from each task in your parallel application. The application is left in a stopped state upon sampling.

Sample Multiple

Gather and merge multiple stack traces from each task in your parallel application over time. The application is left in a stopped state upon sampling.

Sample Options

STAT has several options for stack trace sampling (Figure 6-3).

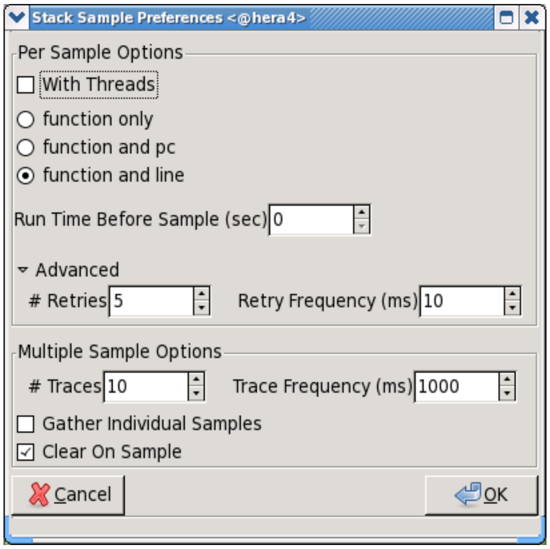

Figure 6-3. The STATGUI operation toolbar.

These options are defined as follows:

With Threads

Sample helper threads in addition to the main thread.

function only | function and pc | function and line

Sample traces with function name only, or function name with the CPU program counter, or function name with the source file and line number.

Run Time Before Sample

Resume the application and let it run for the specified amount of time before gathering the sample

 # Retries/Retry Frequency (Advanced)

Sometimes a process may be in a state (i.e., function prologue or epilogue) such that a complete stack trace may not be obtainable. This option controls how many times to retry sampling and how often to wait between retries to try and get a complete trace.

 # Traces/Trace Frequency

When sampling multiple traces over time, these options specify how many traces to gather per process and how long to wait between samples.

Gather Individual Samples

When sampling multiple traces over time, this option enables STAT to gather all of the intermediate 2D prefix trees in addition to the fully merged 3D prefix tree.

Clear On Sample

When sampling multiple traces over time, STAT accumulates the traces that are gathered. This option determines whether to clear the accumulated traces when gathering additional traces.

Equivalence Classes and Subset Debugging

STATGUI can also serve as an interface to attach a full-featured debugger such as TotalView or DDT to a subset of application tasks. This interface can be accessed through the "identify equivalence classes" **Eq C** button, which will pop up the equivalence classes window (Figure 6-4). You can then select a single representative, all, or none of an equivalence classes' tasks to form a subset of tasks. The **Attach to Subset** buttons will launch the specified debugger and attach to the subset of tasks (note, this detaches STAT from the application). The **Debugger Options** button allows you to modify the debugger path.

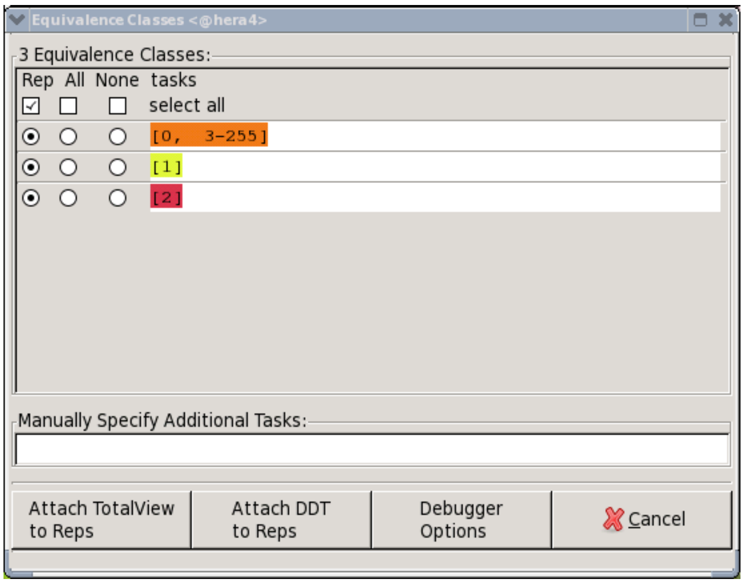

Figure 6-4. The equivalence classes window. The colored task lists correspond to the nodes in the prefix tree.

Availability

The STAT GUI is available on all Peloton and TLCC systems (i.e., Opteron x86_64 machines) and BlueGene systems in **/usr/local/bin/STATGUI**. Man pages are also available (**man STATGUI**).

Chapter 7. Setting STAT Preferences and Options

Preference Files

Several files can influence how STAT runs. The first such file is $prefix/share/nodes.txt, which specifies a list of hostnames, one hostname per line, on which to launch MRNet communication processes. This file is designed to be shared by all users and should point to shared resources that all users have remote shell access to, such as login nodes. Note that STAT will test access to a node before trying to launch communication processes, so it is OK to list nodes that may be down or unaccessible.

STAT GUI preferences can be set with an installation specific or user specific .STATrc file. The installation specific file should be placed in $prefix/share/.STATrc, while the user specific file should be placed in $HOME/.STATrc. Options specified in the user's .STATrc file will always take precedence over the STAT installation's .STATrc file. Each .STATrc file specifies one option per line of the format:

Option = Value

Here is a list of options:

Remote Host = `hostname`

: Sets the default remote host to `hostname` to search for the job launcher process.

Remote Host Shell = `rsh|ssh`

: Sets the default remote host shell to `rsh` or `ssh` to get a process listing on remote hosts.

Job Launcher = `exe`

: Sets the default job launcher for filtering the process listing to `exe` (i.e., mpirun or srun).

Tool Daemon Path = `path`

: Use the STAT deamon executable installed in `path` instead of the default.

Filter Path = `path`

: Use the STAT filter shared object installed in `path` instead of the default.

Communication Nodes = `nodelist`

: Use the nodes listed in `nodelist` for MRNet communication processes.

Communication Processes per Node = `count`

: Launch no more than `count` MRNet communication processes per node.

Retries = `count`

: Attempt `count` retries to try to obtain a complete stack trace.

Retry Frequency (ms) = `count`

: Let the process run `count` milliseconds between retries.

With Threads = `true|false`

: Controls whether to gather stack traces from threads.

Sample Type = `function only|function and pc|function and line`

: Controls the granularity of the stack traces gathered.

DDT Path = *path*

Use the DDT executable installed in *path* for subset debugging.

TotalView Path = *path*

Use the TotalView executable installed in *path* for subset debugging.

Log Dir = *directory*

Write STAT debug logs to *directory*.

Log Frontend = *true|false*

Controls whether to enable debug logging of the STAT frontend.

Log Backend = *true|false*

Controls whether to enable debug logging of the STAT backend.

Loading and Saving Preferences

Options from a STAT session can be saved to a preferences file that can be loaded on subsequent sessions. This can be accessed through the **File** -> **Load Preferences** and **File** -> **Save Preferences** menu items.

Environment Variables

Several environment variables influence STAT and its dependent packages.

STAT_CONNECTION_TIMEOUT=*time*

Wait *time* seconds for daemons to connect to MRNet. Upon timeout, run with the available subset.

STAT_DAEMON=*path*

Use the STAT daemon located at *path* instead of the default.

STAT_FILTER=*path*

Use the STAT filter shared object located at *path* instead of the default.

STAT_MRNET_OUTPUT_LEVEL=*level*

Enable MRNet debug logging at *level* (0-5).

STAT_OUTPUT_REDIRECT_DIR=*directory*

Redirect stdout and stderr to a hostname specific files in *directory*.

STAT_SW_DEBUG_LOG_DIR=*directory*

Enable StackWalker debug logging to a hostname specific files in *directory*.

PROCS_PER_NODE=*count*

(Cray XT only) Specifies that there are *count* processes per compute node.

NODELIST_PATH=*path*

(Cray XT only) The file *path* contains a list of nodes that the application is running on.

Chapter 8. Tips and Tricks Using STAT

Using STAT with IO Watchdog and SLURM

STAT can be used in conjunction with the IO Watchdog[1] utility, which monitors application output to detect hangs. To enable STAT with the IO Watchdog, add the following to the file $HOME/.io-watchdogrc

```
search /usr/local/tools/io-watchdog/actions
timeout = 20m
actions = STAT, kill
```

You will then need to run your application with the --io-watchdog **srun** option:

```
% srun --io-watchdog mpi_application
```

When STAT is invoked, it will create a STAT_results directory in the current working directory, as it would in a typical STAT run. The outputted .dot files can then be viewed with **STATview**. For more details about using IO Watchdog, refer to the IO Watchdog README file in /usr/local/tools/io-watchdog/README.

Running STAT in a Batch Script

A good way to run STAT is at the end of a batch script. For example, if an application is estimated to take 10 hours to run and 12 hours are allocated, then you may consider your application hung if its still running up to the 12th hour. In such a situation, one may choose to run STAT in the last 10 minutes of the allocation to get diagnostic information about the job.

The following example script demonstrates how one might setup STAT to catch a hung job in a batch script.

```
#!/bin/sh

# perform your batch script prologue/setup here

stat_wait_time_minutes=120
application_exited=0

#run the application and get the launcher PID
srun mpi_ringtopo &
pid=$!

# periodically check for application exit
for i in `seq ${stat_wait_time_minutes}`
do
    sleep 60
    ps -p ${pid}
    if test $? -eq 1
    then
        # the application exited, so we're done!
        application_exited=1
        break
    fi
done

# if the application is still running then invoke STAT
if test ${application_exited} -eq 0
```

```
then
    /usr/global/tools/stat/chaos_4_x86_64_ib/stat-test/bin/STATscript -c ${pid}
    waitpid ${pid} # alternatively you may want to `kill -TERM ${pid}`
fi

# perform your batch script epilogue/cleanup here
```

Within the for loop, the script will check every minute (sleep for 60 seconds between checks) to see if the application is still running by running 'ps' on the PID of the job launcher. If the application has exited, the script will break from the loop and perform any remaining operations in the batch script. If the wait time, 120 minutes in this example, expires then STAT will be run to gather stack traces from the application. The wait time should be set such that STAT has enough time to run (i.e., 10 minutes to be safe) within the batch script's allocated time. Note the -c option to STAT gathers a "comprehensive" set of stack traces, with varying levels of detail. After STAT completes, the script then waits for the application to exit. Alternatively, you may want to kill the application if it isn't making any progress.

Notes

1. http://code.google.com/p/io-watchdog/

Chapter 9. Using the STATBench Emulator

Description

The Stack Trace Analysis Tool is a highly scalable, lightweight tool that gathers and merges stack traces from all of the processes of a parallel application. STATBench is a benchmark that can emulate STAT's performance. By utilizing your entire parallel allocation (launching one STATBench daemon emulator per core) and generating artificial stack traces, STATBench is able model STAT's performance using less resources than an actual STAT run requires. With various options, you can also map STATBench to your target machine architecture and target application. After completion, STATBench will create a STAT_results directory in your current working directory. This directory will contain a subdirectory for the current run, with the merged stack traces in DOT format as well as a performance results text file. An example STATBench generated prefix tree emulating 1M (1024*1024) tasks can be seen in Figure 9-1.

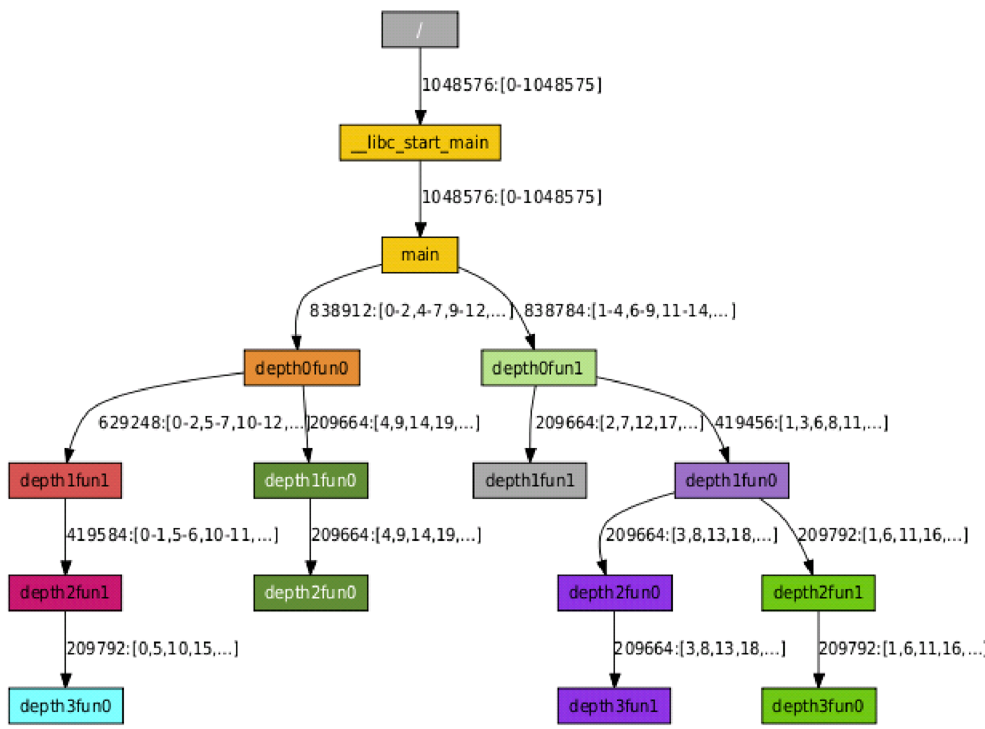

Figure 9-1. A STATBench generated prefix tree emulating over 1 million tasks.

STATBench Options

-a, --autotopo

let STAT automatically create topology.

-f, --fanout `width`

Sets the maximum tree topology fanout to `width`. Specify nodes to launch communications processes on with `--nodes`.

-d, --depth *`depth`*

Sets the tree topology depth to *`depth`*. This option takes precedence over the `--fanout` option. Specify nodes to launch communications processes on with `--nodes`.

-u, --usertopology *`topology`*

Specify the number of communication nodes per layer in the tree topology, separated by dashes, with *`topology`*. This option takes precedence over the `--fanout` and `--depth` options. Specify nodes to launch communications processes on with `--nodes`. Example topologies: 4, 4-16, 5-20-75.

-n, --nodes *`nodelist`*

Use the specified nodes in *`nodelist`*. To be used with `--fanout`, `--depth`, or `--usertopology` options. Example nodes lists: host1; host1,host2; host[1,5-7,9].

-p, --procs *`processes`*

Sets the maximum number of communication processes to be spawned per node to *`processes`*. This should typically be set to the number of CPUs per node.

-D, --daemon *`path`*

Specify the full path *`path`* to the STATBench daemon executable. Use this only if you wish to override the default.

-F, --filter *`path`*

Specify the full path *`path`* to the STATBench filter shared object. Use this only if you wish to override the default.

-t, --traces *`count`*

Gather *`count`* traces per process.

-i, --iters *`count`*

Perform *`count`* gathers.

-n, --numtasks *`count`*

Emulate *`count`* tasks per daemon.

-m, --maxdepth *`depth`*

Generate traces with a maximum depth of *`depth`*.

-b, --branch *`width`*

Generate traces with a max branching factor of *`width`*.

-e, --eqclasses *`count`*

Generate traces within *`count`* equivalence classes.

-l, --log

[*`FE`* | *`BE`* | *`ALL`*]

Enable debug logging of the *`FE`*, *`BE`*, or *`ALL`*.

-L, --logdir `log_directory`

Dump logging output into `log_directory`. To be used with the `--log` option.

STATBench Usage Example

In the simplest form, you can invoke STATBench, from within a parallel allocation, with no arguments. This will run through with the default settings:

```
% STATBench
```

To model your target machine architecture, you can specify the number of tasks to emulate per daemon. For instance if your target machine has 16-way SMP compute nodes:

```
% STATBench --numtasks 16
```

You may also want to model a specific application. For instance, you may have a climate modeling code with 5 distinct task behaviors, or equivalence classes. You can also specify the maximum call depth of your application, the average branching factor from a given function, and the number of distinct traces expected per task:

```
% STATBench --eqclasses 5 --maxdepth 17 --branch 5 --traces 4
```

At larger scales, you may want to employ a more scalable tree topology. For example, if you're running 1024 daemon emulators, you may want to try a fanout of sqrt(1024) = 32. You will need to specify a list of nodes that contains enough processors to accommodate the ceil(1024/32) = 32 communication processes being launched. Be sure that you have login permissions to the specified nodes and that they contain the mrnet_commnode executable and the STAT_FilterDefinitions.so library:

```
% STATBench --fanout 32 atlas[1-4] --procs 8
```

Chapter 10. Troubleshooting Guide

Troubleshooting

stack walks not making it to _start

Processes can be in portions of code from which a debugger cannot walk the stack (i.e., function prologue or epilogue). Try the -r option to enable STAT to let the process run a bit and then retry the stack sample.

/usr/lib/python2.6/site-packages/gtk-2.0/gtk/__init__.py :72: GtkWarning: could not open display

Be sure to enable X-forwarding and to set your $DISPLAY environment variable.

STATview requires gtk

STAT requires the pygtk module to be installed. If it is side-installed, but sure to set your $PYTHONPATH environment variable to the directory containing the pygtk module.

ImportError: No module named STAT

Make sure STAT.py is in PYTHONPATH or bin directory

(ERROR): LaunchMON Engine invocation failed, exiting: No such file or directory

Make sure the **launchmon** executable is in your $PATH or set the $LMON_LAUNCHMON_ENGINE_PATH engine path to the full path to the executable.

OptionParsing (ERROR): unknown launcher: a.out

You need to attach to your **mpirun** or equivalent parallel job launch process.

OptionParsing (ERROR): the path[/usr/local/bin/STATD] does not exit.

STAT looks for its components in the configured $prefix. Be sure to run 'make install' or set STAT_DAEMON_PATH to the full path to the **STATD** executable.

LaunchMON prints a usage message.

This is typically a mismatch in versions of the LaunchMON API and the LaunchMON engine. Make sure to set your LMON_LAUNCHMON_ENGINE_PATH enviornment variable to the full path to the appropriate **launchmon** executable.

(ERROR): accepting a connection with an engine timed out

STAT may need additional time to launch all of its daemons. You may need to set your $LMON_FE_ENGINE_TIMEOUT to a larger value, such as 600.

Bibliography

Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz, "Scalable Temporal Order Analysis for Large Scale Debugging," *Supercomputing 2009,* Portland, Oregon, November 2009.

Gregory L. Lee, Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit, "Lessons Learned at 208K: Towards Debugging Millions of Cores," *Supercomputing 2008,* Austin, Texas, November 2008.

Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz, "Overcoming Scalability Challenges for Tool Daemon Launching," *37th Internation Conference on Parallel Processing (ICPP-08),* Portland, Oregon, September, 2008.

Gregory L. Lee, Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Barton P. Miller, and Martin Schulz, "Benchmarking the Stack Trace Analysis Tool for BlueGene/L," *International Conference on Parallel Computing (Parco) 2007*, Aachen and Julich, Germany, September 2007.

Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz, "Stack Trace Analysis for Large Scale Applications," *International Parallel & Distributed Processing Symposium,* Long Beach, California, March 2007.

Notes

1. ftp://ftp.cs.wisc.edu/paradyn/papers/Miller09ScalableDebugging.pdf
2. ftp://ftp.cs.wisc.edu/paradyn/papers/Lee08ScalingSTAT.pdf
3. ftp://ftp.cs.wisc.edu/paradyn/papers/Ahn08LaunchMON.pdf
4. ftp://ftp.cs.wisc.edu/paradyn/papers/Lee07STATBench.pdf
5. ftp://ftp.cs.wisc.edu/paradyn/papers/Arnold06STAT.pdf