STAT: the Stack Trace Analysis Tool

Gregory L. Lee
Dorian C. Arnold
Dong H. Ahn
Bronis R. de Supinski
Matthew P. LeGendre
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 Matthew P. LeGendre
by Barton P. Miller
by Martin Schulz
Table of Contents

Disclaimer .......................................................................................................................v
Auspice ...................................................................................................................v
License ....................................................................................................................v
1. Introduction ................................................................................................................1
2. Overview .....................................................................................................................3
3. Changelog....................................................................................................................7
   STAT version 2.1 ....................................................................................................7
   STAT version 2.0 ....................................................................................................7
4. Installing STAT ..........................................................................................................9
   Dependent Packages ....................................................................................................9
   Installation ....................................................................................................................9
5. Using the stat-cl Command....................................................................................11
   Description ...........................................................................................................11
   stat-cl Options ......................................................................................................11
   STAT Usage Example ............................................................................................13
6. Using the stat-view GUI .........................................................................................15
   Description ...........................................................................................................15
   The stat-view Node Menu .......................................................................................15
   The stat-view Toolbar .........................................................................................17
7. Using the stat-gui GUI............................................................................................21
   Description ...........................................................................................................21
   The stat-gui GUI Toolbar ....................................................................................21
   Sample Options ...................................................................................................22
   Process Table ........................................................................................................24
   Equivalence Classes and Subset Debugging ...................................................24
   Availability ...........................................................................................................25
8. Setting STAT Preferences and Options ...............................................................27
   Preference Files ....................................................................................................27
   Loading and Saving Preferences .......................................................................29
   Environment Variables .......................................................................................29
9. Tips and Tricks Using STAT ..................................................................................33
   Using STAT with IO Watchdog and SLURM ..................................................33
   Running STAT in a Batch Script ........................................................................33
10. Using the stat-bench Emulator ............................................................................35
    Description ...........................................................................................................35
    stat-bench Options ..............................................................................................35
    stat-bench Usage Example ....................................................................................37
11. Troubleshooting Guide.........................................................................................39
    Troubleshooting ...................................................................................................39
Bibliography .................................................................................................................41
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

Copyright (c) 2007-2014, Lawrence Livermore National Security, LLC.
Produced at the Lawrence Livermore National Laboratory
Written by Gregory Lee [lee218@llnl.gov], Dorian Arnold, Matthew LeGendre, Dong Ahn, Bronis de Supinski, Barton Miller, and Martin Schulz.
LLNL-CODE-624152.

All rights reserved.

This file is part of STAT. For details, see http://www.paradyn.org/STAT/STAT.html. 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 is developed as a collaboration between the Lawrence Livermore National Laboratory, the University of Wisconsin, and the University of New Mexico. 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 or DDT for more in-depth analysis.

STAT has been ported to several platforms, including Linux clusters, IBM’s Bluegene/L, Bluegene/P, and Bluegene/Q machines, and Cray 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 1,000,000 MPI tasks and its logarithmic scaling characteristics position it well for even larger systems.

Notes

Chapter 1. Introduction
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)](image)

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.
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.

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 2. Overview
Chapter 3. Changelog

STAT version 2.1

- Added prototype DysectAPI
- Added prototype temporal ordering capability
- Improved Python wrapping, including new python-script command and a Python-based test script to run through STAT functionality
- Modified cut/hide interface to allow for user-defined programming models
- Modified dot file naming convention for easier sequencing. Also allow specification of dot filename on gather.
- New process table interface (via File->Properties)
- Added CP location policy
- Improved BG/Q support, including use of StackWalker timeouts.
- Improved FGFS support, including adding BG/Q support and re-enabling MRNet failure recovery when FGFS is in use
- Better labeling of error tasks in stack traces
- As always, numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)

STAT version 2.0

- The capitalized STAT commands have been deprecated in favor of all lower-case commands. The STAT command is now stat-cl, STATGUI is now stat-gui, STATview is now stat-view, and STATBench is now stat-bench.
- Added optional build with Fast Global File Status plus target application binary file broadcasting
- Added Python-script-level debugging (Python must be built with -g and preferably -O0)
- Added ability to manually specify serial processes to attach to
- Added count + representative level of detail
- Added join equivalence class GUI feature
- Added cut text GUI feature
- Added GUI preferences menu item
- (old) DynInst support has been removed. STAT now strictly requires the Stackwalker component of DynInst.
- Graphlib 2.0 required (removed support for 1.X)
- MRNet version 3 or greater required (removed support for 1.X and 2.X)
- Added ability to manually specify list of processes
- Numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)
Chapter 3. Changelog
Chapter 4. Installing STAT

Dependent Packages

STAT has several dependencies

Table 4-1. STAT Dependent Packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Package Web Page</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphlib version 2.0 or greater</td>
<td><a href="https://github.com/lee218llnl/STAT/">https://github.com/lee218llnl/STAT/</a></td>
<td>Graph creation, merging, and export</td>
</tr>
<tr>
<td>Launchmon</td>
<td><a href="http://sourceforge.net/projects/launchmon/">http://sourceforge.net/projects/launchmon/</a></td>
<td>Scalable daemon co-location</td>
</tr>
<tr>
<td>Libdwarf</td>
<td><a href="http://reality.sgiweb.org/davea/dwarf.html">http://reality.sgiweb.org/davea/dwarf.html</a></td>
<td>Debug information parsing (Required by StackWalker)</td>
</tr>
<tr>
<td>MRNet version 3.0 or greater</td>
<td><a href="http://www.paradyn.org/mrnet/">http://www.paradyn.org/mrnet/</a></td>
<td>Scalable multicast and reduction network</td>
</tr>
</tbody>
</table>

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.

STAT can also be optionally built with the Fast Global File Status (FGFS)\(^4\) library. This library helps STAT identify when a file (target binary) resides on a shared file system that may become a bottleneck if all STAT daemons try to access that file at the same time. If so, STAT will access the file from the STAT frontend and distribute its contents to the daemons via the MRNet communication tree. The two necessary components of FGFS can be downloaded from https://github.com/dongahn/MountPointAttributes and https://github.com/dongahn/FastGlobalFileStatus

Installation

A script that downloads and install STAT and all its dependencies for BlueGene/Q has been created and can be downloaded at https://github.com/lee218llnl/STAT/blob/master/script-stat-install or can be found in the top-level directory of the STAT source. Note this script was developed for BlueGene/Q systems, but can be used as guidance for other systems as well.

When building STAT itself, 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 include 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 creates wrapper scripts for the stat-cl and stat-gui commands. These wrappers set appropriate paths for the launchmon and mrnet_commnode executables, based on the the --with-launchmon and --with-mrnet configure options, thus it is important to specify both of these even if they share a prefix.
Chapter 4. Installing STAT

STAT will try to build the GUI by default. If you need to modify your PYTHON-PATH environment variable to search for side installed site-packages, you can do this by specifying STAT_PYTHONPATH=path during configure. This will add the appropriate directory to the $PYTHONPATH environment variable within the stat-gui script. To disable the building of the GUI, use the --enable-gui=no configure option.

On BlueGene systems, be sure to configure --with-bluegene. This will enable the BGL macro for BlueGene specific compilation. It is important to note that on BlueGene systems, you may need to use an alternate hostname for the front-end node in order to get MRNet to bind to the appropriate network interface that can communicate with the I/O nodes. By default, STAT will append "-io" to the hostname. Alternatively, you can specify the hostname with the STAT_FE_HOSTNAME environment variable. A script that downloads and install STAT and all its dependencies for BlueGene/Q has been created and can be downloaded at https://github.com/lee218llnl/STAT/blob/master/script-stat-install or can be found in the top-level directory of the STAT source. Due to system configurations, this may not completely work out of the box, but it provides good guidance on the steps required.

To compile on Cray systems, specify --with-cray-xt. An example configure line for Cray:

```
./configure --with-launchmon=/tmp/work/lee218/install \
--with-mrnet=/tmp/work/lee218/install \
--with-graphlib=/tmp/work/lee218/install \
--with-stackwalker=/tmp/work/lee218/install \
--with-libdwarf=/tmp/work/lee218/install \
--prefix=/tmp/work/lee218/install --with-cray-xt \nMPICC=cc MPICXX=CC MPIF77=ftn --enable-shared LD=/usr/bin/ld
```

Note that specifying LD=/usr/bin/ld may be required on Cray systems to avoid using the compute node linker. It is also worth noting that Cray includes a build of STAT as part of their system software stack. It is typically installed in /opt/cray/stat and can be loaded via modules.

After running configure 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. Further, the STAT_PREFIX environment variable can be defined to override the hardcoded paths in STAT. STAT will also, by default, add rpaths to dependent libraries. This behavior can be disabled by specifying --with-rpath=no. However, when doing so, you must be sure to set LD_LIBRARY_PATH to point to the directories containing the dependent libraries.

Notes

Chapter 5. Using the stat-cl 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-cl 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. 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. 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].

-\( A, --appnodes \)
  Allow tool communication processes to be co-located on nodes running application processes.

-\( x, --exclusive \)
  Do not use the front-end or back-end nodes for communication processes.

-\( p, --procs \) processes
  Sets the maximum number of communication processes to be spawned per node to processes. This should typically be set to a number less than or equal to the number of CPU cores per node.

-\( j, --jobid \) id
  Append id to the output directory 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 microseconds between sample retries. To be used with the \( --retries \) option.
Chapter 5. Using the stat-cl Command

- **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.

- **U, --countrep**
  Only gather edge labels with the task count and a single representative. This will improve performance at extreme (i.e., over 1 million tasks) scales.

- **w, --withthreads**
  Sample stack traces from helper threads in addition to the main thread.

- **y, --withpython**
  Where applicable, gather Python script level stack traces, rather than show the Python interpreter stack traces. This requires the Python interpreter being debugged to be built with -g and preferably -O0.

- **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.

- **I, --serial arg_list**
  Attach to a list of serial processes. All arguments after -I are interpreted as processes. Namely, arg_list is a white-space-separated list of processes to attach to, where each process is of the form [exe@][hostname:]PID.

- **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**
Enable debug logging of the FE frontend, BE backend, CP communication process, SW Stackwalker, SWERR Stackwalker on error. Multiple log options may be specified (i.e., -l FE -l BE).

-L, --logdir log_directory
Dump logging output into log_directory. To be used with the --log option.

-M, --mrnetprintf
Use MRNet’s printf for STAT debug logging.

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-cl 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-cl -C srun mpi_application arg1 arg2
```

With the -a option (or when automatic topology is set as default), STAT will try to automatically create a scalable topology for large scale jobs. However, if you wish you may manually specify a topology at larger scales. 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 with the --nodes option. 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-cl --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 stat-view.
Chapter 6. Using the stat-view GUI

Description

stat-view (Figure 6-1) is a GUI for viewing STAT-outputted dot files. stat-view 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, stat-view will display the number of tasks in the set and truncate long task lists in the main display with “...” notation. Similarly, long function node label names will be truncated with “...” notation. The truncation length can be modified via the File->Preferences menu (this requires clicking the Layout button to rerender any already loaded graphs). Nodes are colored based on the set of tasks of the incoming edge, providing a visual distinction when different tasks take different branches.

Figure 6-1. A screenshot of the stat-view GUI.

The stat-view 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 6-2). 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.
Chapter 6. Using the stat-view GUI

Figure 6-2. The node pop-up window

The node operations are defined as follows:

**Join Equivalence Class**

- collapses all of the descendent nodes with the same equivalence class into the current node and renders in a new tab.

**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 6-3) 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.
Chapter 6. Using the stat-view GUI

Figure 6-3. The source view window. The colored arrows correspond to the nodes in the call prefix tree.

The stat-view Toolbar

The main window also has several tree manipulation options (Figure 6-4). 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 6-4. The stat-view 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 stat-view 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

Layout
- Reset the layout of the current graph and open in a new tab. This is useful for compacting wide trees after performing some pruning operations.

Cut
- This feature (Figure 6-5) allows you to collapse the prefix tree below the implementation frames for various programming models. For
instance, a user may wish to hide all calls that happen within the MPI library. The programming models may be entered in a configuration file or added by the user. STATview looks for configuration files in $prefix/etc/STAT/STATview_models.conf and in $HOME/.STATview_models.conf. Programming models are specified as regular expressions, using Python’s `re` module syntax, and the `re.search` function is used in favor of `re.match`.

![ STATview XT programming model cutting interface.](image)

**Figure 6-5.** The stat-view programming model cutting interface.

- **[Cut] MPI**
  Collapse the MPI implementation frames below the MPI function call.

- **[Cut] Pthreads**
  Collapse the Pthread implementation frames below the Pthread function call.

- **Join**
  Join consecutive nodes of the same equivalence class into a single node and render in a new tab. This is useful for condensing long call sequences.

- **[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.
Chapter 6. Using the stat-view GUI

[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.

Search
Search for call paths containing specified text, taken by specified tasks, or from specified hosts. Search text may be a regular expression, using the syntax described in http://docs.python.org/library/re.html.

[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 7. Using the stat-gui GUI

Description

STAT includes a graphical user interface (GUI) to run STAT and to visualize STAT’s outputted call prefix trees (Figure 7-1). 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.

The stat-gui GUI Toolbar

In addition to the operations provided by stat-view, stat-gui provides a toolbar (Figure 7-2) to control STAT’s operation.
Chapter 7. Using the stat-gui GUI

Figure 7-2. The STAT GUI toolbar.

Attach
Attach to your application and gather an initial sample.

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

Detach
Detach from your 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 after 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 after sampling.

Sample Options
STAT has several options for stack trace sampling (Figure 7-3).
Chapter 7. Using the stat-gui GUI

Figure 7-3. The stat-gui operation toolbar.

These options are defined as follows:

**With Threads**
Sample helper threads in addition to the main thread.

**Gather Python Traces**
Where applicable, gather Python script level stack traces, rather than show the Python interpreter stack traces. This requires the Python interpreter being debugged to be built with -g and preferably -O0.

- `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.

- `full list`
- `count and representative`

Sample traces with the full task list or just the count and a single representative. When gathering the count and representative, you can actually query an individual STAT graph node (through the left-click menu) for the full edge label, as long as the STAT session is still attached.

**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 (in microseconds) 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. The traces will be displayed in individual tabs.
Chapter 7. Using the stat-gui GUI

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.

Process Table

The application process table can be accessed through the stat-gui File->Properties menu item. This window (Figure 7-4) lists the properties of the application, including the number of nodes, processes, the job launcher host and PID and a 4-tuple list of application process rank, host, PID, and executable. The executable path in the 4-tuple is an index into the executable list at the top of the window in order to reduce duplication of text. The 4-tuple process table list can be filtered by ranks or hosts.

![Figure 7-4. The properties window shows the application properties and lists the individual application processes.](image)

Equivalence Classes and Subset Debugging

stat-gui 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 7-5). 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 7-5. The equivalence classes window. The colored task lists correspond to the nodes in the prefix tree.

**Availability**

The STAT GUI is available on all TLCC systems (i.e., x86_64 machines) and Blue-Gene systems in `/usr/local/bin/stat-gui`. Man pages are also available (`man stat-gui`).
Chapter 7. Using the stat-gui GUI
Chapter 8. Setting STAT Preferences and Options

Preference Files

Several files can influence how STAT runs. The first such file is $prefix/etc/STAT/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 by default STAT will not test access to a node before trying to launch communication processes. If the STAT_CHECK_NODE_ACCESS environment variable is set to any value, then STAT will try to run (via remote shell) a simple test to see if the node is accessible before adding it to the MRNet tree. Also note that nodes.txt will not be used if the -A or "Share App Nodes" option is enabled.

STAT GUI preferences can be set with an installation specific STAT.conf or user specific .STATrc file. The installation specific file should be placed in $prefix/etc/STAT/STAT.conf, 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 preference 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 = regex

Sets the default regular expression to regex (i.e., "mpirun|srun") for filtering the process listing for the job launcher process.

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.

Topology Type = automatic|depth|max fanout|custom

Use the specified topology type when building the MRNet communication tree. The automatic topology is typically recommended and set by default.

Topology = topology

Use topology for the specific topology configuration. This should be used with the Topology Type option. Refer to the stat-cl options to see valid Topology specifications for a given Topology Type.

Communication Nodes = nodelist

Use the nodes listed in nodelist for MRNet communication processes.

Check Node Access = true|false

Controls whether to check access to a node before trying to launch MRNet communication processes on it.
Chapter 8. Setting STAT Preferences and Options

CP policy = none|share app nodes|exclusive

Controls where to launch communication processes. When set to share app nodes, they will be launched on nodes running application processes. On BlueGene systems, this will actually place them on the I/O nodes, and requires users to be able to access the I/O nodes via a remote shell. When set to exclusive, then the communication processes will only be run on specified nodes that do not run other STAT tool processes (e.g., the STAT frontend and the back-end daemons).

Communication Processes per Node = count

Launch no more than count MRNet communication processes per node.

Num Traces = count

Gather count stack traces when sampling multiple.

Trace Frequency (ms) = count

Let the process run count milliseconds between multiple samples.

Num Retries = count

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

Retry Frequency (ms) = count

Let the process run count milliseconds between stack sample retries.

With Threads = true|false

Controls whether to gather stack traces from threads.

Gather Python Traces = true|false

Controls whether to gather Python script level stack traces, rather than show the Python interpreter stack traces.

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

Controls the granularity of the nodes in the gathered stack traces.

Edge Type = full list|count and representative

Controls the granularity of the edges in the gathered stack traces.

DDT Path = path

Use the DDT executable installed in path for subset debugging.

DDT LaunchMON Prefix = path

Use the LaunchMON installation in path for improved DDT subset attaching, otherwise attach via hostname:PID pairs.

TotalView Path = path

Use the TotalView executable installed in path for subset debugging.

Additional Debugger Args = args

Add args to the argument list when launching TotalView or DDT.

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.
Chapter 8. Setting STAT Preferences and Options

Log CP = true/false
Controls whether to enable debug logging of the STAT communication processes.

Log SW = true/false
Controls whether to enable debug logging of Stackwalker by the STAT backend.

Log SWERR = true/false
Controls whether to enable debug logging of Stackwalker by the STAT backend when a Stackwalker error is detected.

Use MRNet Printf = true/false
Controls whether to use MRNet's printf when writing debug logs. This is helpful to correlate timing between STAT log messages and MRNet debug log messages, when MRNet logging is being logged (via the STAT_MRNENT_OUTPUT_LEVEL environment variable).

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. Note that dependent package environment variables are prefixed with "STAT_" to avoid conflict with other tools using that package. The STAT process will then set the appropriate (i.e., without "STAT_") environment variable to pass the value to the dependent package.

STAT_PREFIX=directory
Use directory as the installation prefix instead of the compile-time STAT_PREFIX macro when looking for STAT components and configuration files.

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

STAT_DAEMON_PATH=path
Use the STAT daemon executable path instead of the default. path must be set to the full path of the STATD executable.

STAT_FILTER_PATH=path
Use the STAT filter shared object path instead of the default. path must be set to the full path of the STAT_FilterDefinitions.so shared object file.

STAT_FGFS_FILTER_PATH=path
Use the STAT FGFS filter shared object path instead of the default. path must be set to the full path of the STAT_FilterDefinitions.so shared object file.

STAT_MRNENT_OUTPUT_LEVEL=level
Enable MRNet debug logging at level (0-5).
Chapter 8. Setting STAT Preferences and Options

STAT_MRNET_PORT_BASE=port
Set the MRNet base port number to port.

STAT_MRNET_STARTUP_TIMEOUT=seconds
Set the MRNet connection timeout to seconds.

STAT_CONNECT_TIMEOUT=seconds
Set the STAT connection timeout to seconds, after which STAT will try to continue with any subset of daemons that have connected.

STAT_MRNET_DEBUG_LOG_DIRECTORY=directory
Write MRNet debug log files to directory.

STAT_OUTPUT_REDIRECT_DIR=directory
Redirect stdout and stderr to a set of hostname specific files in directory.

STAT_MRNET_COMM_PATH=path
Use the mrnet_commnode executable path. path must be set to the full path of the mrnet_commnode executable. (Deprecated along with MRNet’s MRN_COMM_PATH)

STAT_MRNET_COMM_PATH=path
Use the mrnet_commnode executable path. path must be set to the full path of the mrnet_commnode executable.

STAT_XPLAT_RSH=path
Use the remote shell path for launching mrnet_commnode processes.

STAT_PROCS_PER_NODE=count
Allow up to count communication processes to be launched per node.

STAT_FE_HOSTNAME=value
Set the STAT Front End hostname to value. This may be necessary for example on BlueGene systems to use the proper network interface for the I/O nodes to connect back to.

STAT_CHECK_NODE_ACCESS=value
Set to any value to have STAT check user access to any specified nodes before launching communication processes.

STAT_GROUP_OPS=value
Set to any value to enable Stackwalker’s group operations. Group operations may help with performance when a single daemon needs to manage a large number of target processes. This is on by default on BG/Q systems.

STAT_LMON_PREFIX=path
Sets the LaunchMON installation prefix to path.

STAT_LMON_LAUNCHMON_ENGINE_PATH=path
Use the launchmon executable path. path must be set to the full path of the launchmon executable.

STAT_LMON_REMOTE_LOGIN=command
Use the remote shell command for LaunchMON remote debugging.

STAT_LMON_DEBUG_BES=value
Launch the backends under a debugger’s control if value is set (must be enabled in LaunchMON configuration).
STAT_USAGE_LOG=path

Record usage of STAT in the file located in path. path must be writeable by user.

STAT_GROUP_OPS=value

If this environment variable is set on the daemons, STAT will use Stack-walker’s group operations instead of operating on individual processes. This is set by default on BG/Q systems.
Chapter 9. Tips and Tricks Using STAT

Using STAT with IO Watchdog and SLURM

STAT can be used in conjunction with the IO Watchdog 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 stat-view. 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 it is 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/local/bin/stat-cl -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

Chapter 10. Using the stat-bench 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. stat-bench is a benchmark that can emulate STAT’s performance. By utilizing your entire parallel allocation (launching one stat-bench daemon emulator per core) and generating artificial stack traces, stat-bench is able to model STAT’s performance using less resources than an actual STAT run requires. With various options, you can also map stat-bench to your target machine architecture and target application.

After completion, stat-bench 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 stat-bench generated prefix tree emulating 1M (1024*1024) tasks can be seen in Figure 10-1.

stat-bench 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].

-A, --appnodes
Allow tool communication processes to be co-located on nodes running application processes.

-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 STATBenchD daemon executable. Use this only if you wish to override the default.

-F, --filter path
Specify the full path path to the stat-bench 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.

-U, --countrep
Only gather edge labels with the task count and a single representative.

-l, --log
[FE | BE | CP]
Enable debug logging of the FE frontend, BE backend, or CP communication process. Multiple log options may be specified (i.e., -l FE -l BE).
Chapter 10. Using the stat-bench Emulator

-L, --logdir log_directory
    Dump logging output into log_directory. To be used with the --log option.

-M, --mrnetprintf
    Use MRNet’s printf for STAT debug logging.

stat-bench Usage Example

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

% stat-bench

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:

% stat-bench --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:

% stat-bench --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 \(\text{ceil}(1024/32) = 32\) 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-bench --fanout 32 --nodes atlas[1-4] --procs 8
Chapter 11. 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.

stack walks with line number information returning ??
Stack traces with line number information requires your code to be compiled with debug information (i.e., with the -g flag).

/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 to run ‘make install’ to install STAT.py in the lib/python[version]/site-packages directory or set your $PYTHONPATH environment variable to the directory containing STAT.py

(ERROR): LaunchMON Engine invocation failed, exiting: No such file or directory
Make sure the launchmon executable is in your $PATH or set the $STAT_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 $STAT_LMON_LAUNCHMON_ENGINE_PATH environment 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


Notes
