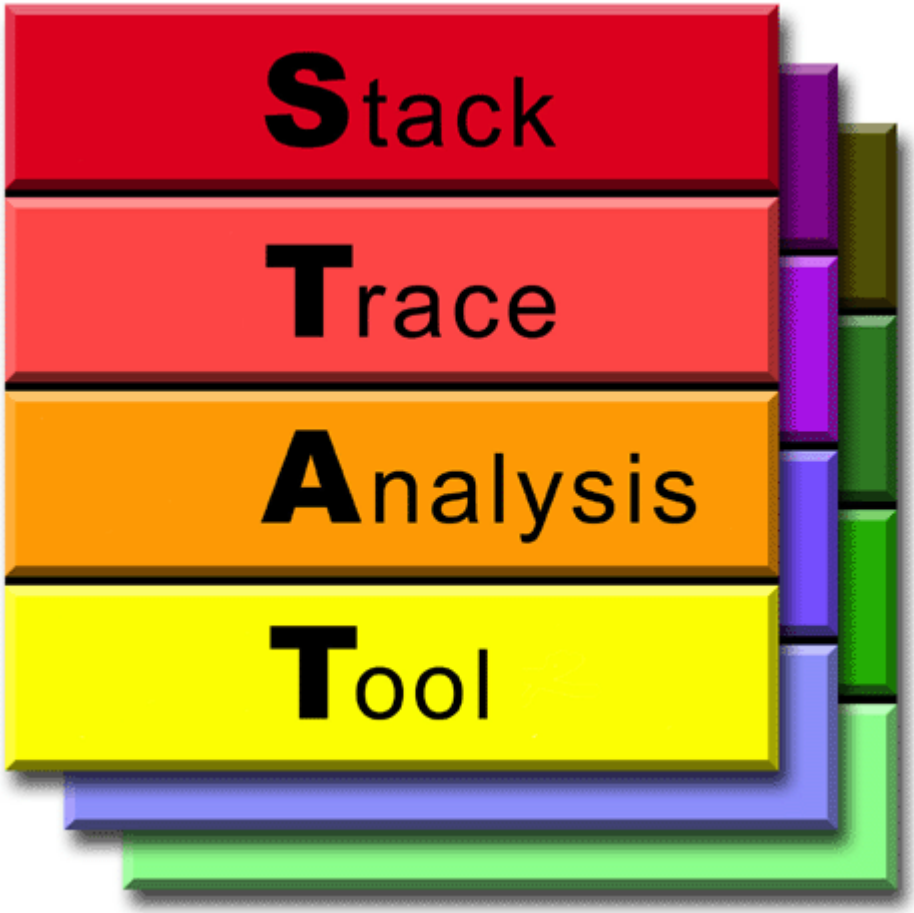
STAT: the Stack Trace Analysis Tool



Gregory L. Lee

Dorian C. Arnold

Dong H. Ahn

Bronis R. de Supinski

Nicklas B. Jensen

Sven Karlsson

Matthew P. LeGendre

Barton P. Miller

Niklas Q. Nielsen

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 Nicklas B. Jensen

by Sven Karlsson

by Matthew P. LeGendre

by Barton P. Miller

by Niklas Q. Nielsen

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

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, the University of New Mexico, and the Denmark Technical University. 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 x86, Power, and Arm 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 can also use cuda-gdb for its backend, enabling it to gather GPU stack traces in CUDA codes. STAT has already demonstrated scalability over 3,000,000 MPI tasks and its logarithmic scaling characteristics position it well for even larger systems.

Notes

1. http://www.roguewave.com/products-services/totalview
2. http://www.allinea.com

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.

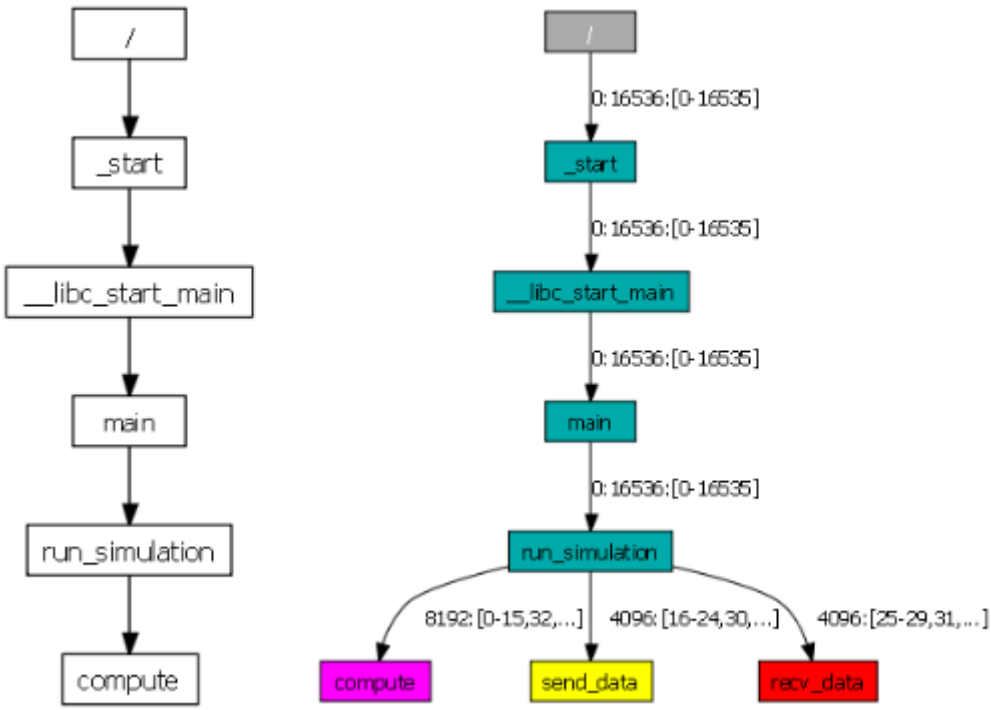


Figure 2-1. A single stack trace (left) and a STAT merged call prefix tree (right)

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.

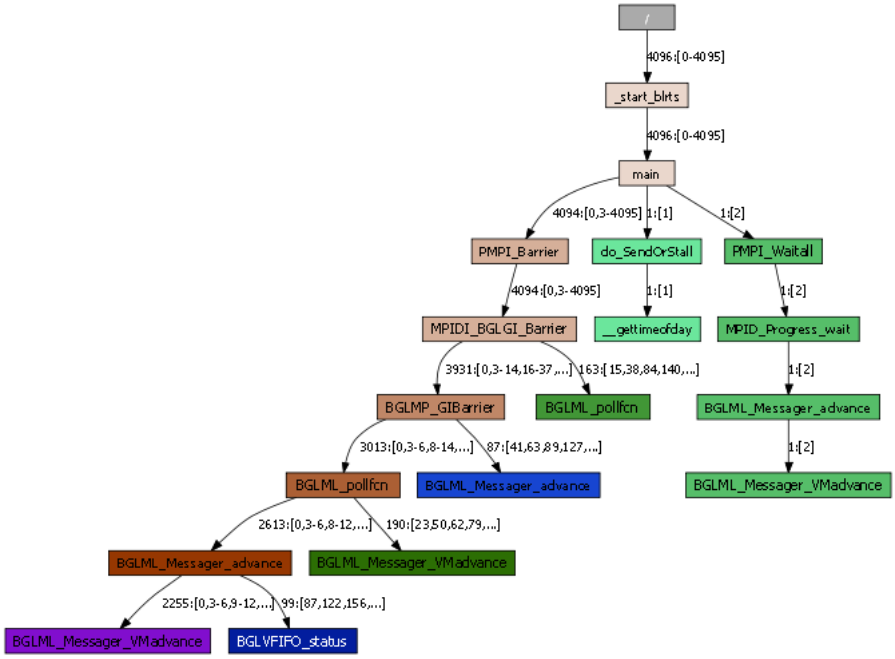


Figure 2-2. A 2D spatial call prefix tree

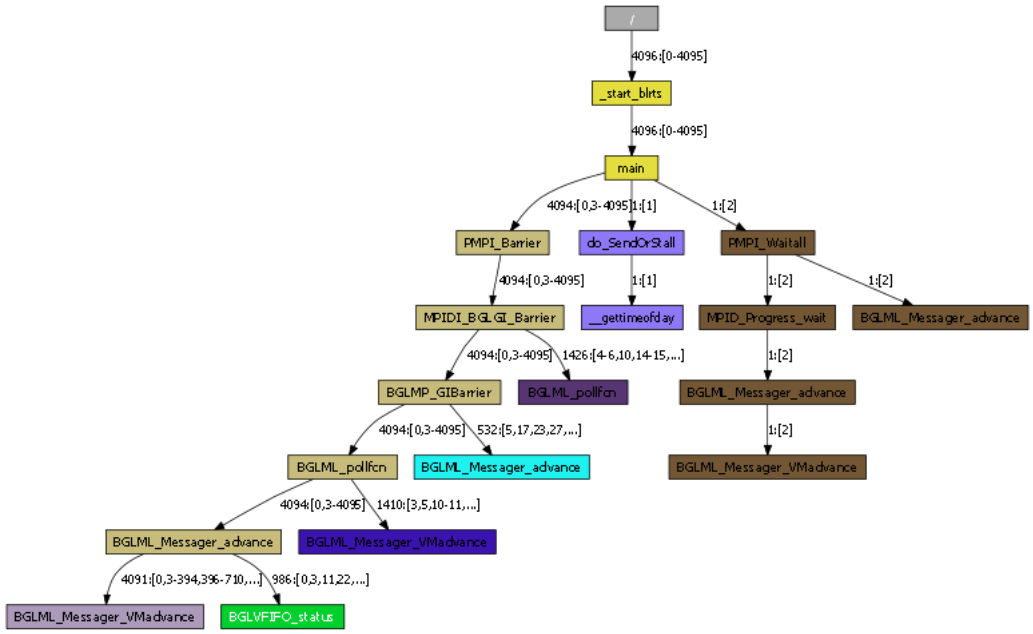


Figure 2-3. A 3D spatial call prefix tree

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.

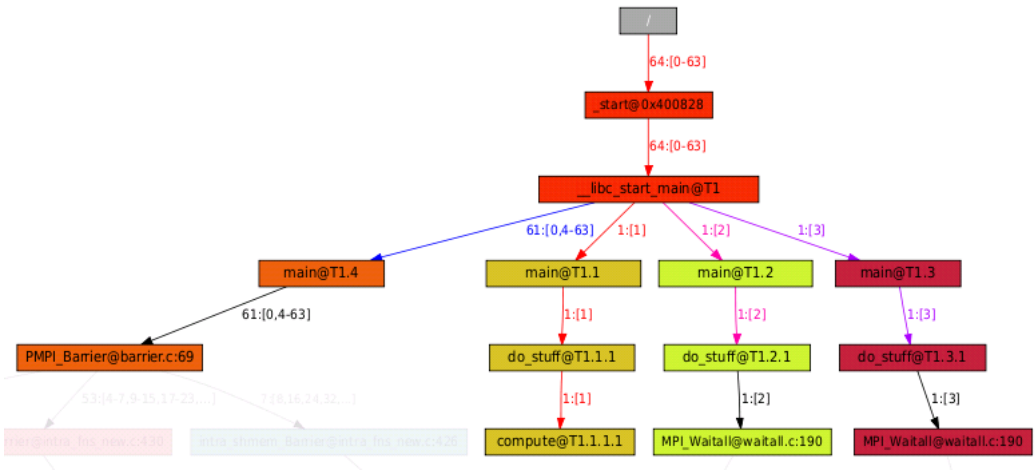


Figure 2-4. STAT's temporal ordering analysis engine indicates that task 1 has made the least progress. In this example, task 1 is stuck in a compute cycle, while the other tasks are blocked in MPI communication, waiting for task 1.

Chapter 3. Changelog

stat version 4.0.3

- DynInst 10.2 support
- as always, numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)

stat version 4.0.2

- DynInst 10.X support
- Core file merging enhancements
- as always, numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)

stat version 4.0.1

- Python 3 and xdot 0.9 support added
- as always, numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)

stat version 4.0.0

- Added GDB and Cuda GDB backend support
- Core file merging enhancements
- GUI enhancements
- ability for Dysect API to dump lightweight core via callpath
- as always, numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)

stat version 3.0.1

- Dyninst 9.3.0 support
- added ability to launch multiple daemons per node
- as always, numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)

stat version 3.0

- added OpenMP OMPD support and ehanced thread display
- added data analytics to DySectAPI
- added DySectAPI probe tree visualizer
- fixes to FGFS and file broadcasting
- STAT now requires graphlib 3.0 to support generic edge labels
- as always, numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)

stat version 2.2

- most edits for this release were bug fixes and enhancements to the prototype DySectAPI (refer to ChangeLog file in top-level directory for details of changes)
- DySectAPI session added to STAT GUI
- added module offset granularity
- cleaner daemon crash handling
- build system fixes
- as always, numerous other bug fixes and minor enhancements (refer to ChangeLog file in top-level directory)

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 lowercase 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 preferrably -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 4. Installing STAT

Dependent Packages

STAT has several dependencies

Table 4-1. STAT Dependent Packages

Package Package Web Page	What It Does
Graphlib version 3.0 or greater https://github.com/LLNL/graphlib/	Graph creation, merging, and export
Launchmon https://github.com/LLNL/LaunchMON	Scalable daemon co-location
Libdwarf https://www.prevanders.net/dwarf.html	Debug information parsing (Required by StackWalker)
MRNet version 3.0 or greater https://github.com/dyninst/mrnet	Scalable multicast and reduction network
StackWalker https://github.com/dyninst/dyninst	Lightweight stack trace sampling

In addition, STAT requires Python[1] and the STAT GUI requires PyGTK[2], both of which are commonly preinstalled with many Linux operating systems. STAT can be built with Python 2.X and PyGTK 2.X. However, starting with STAT version 4.0.1, STAT can optionally be built with Python 3.X. The use of Python 3.X also requires PyGTK version 3.X and requires you to manually install the xdot python package[3]. STAT also requires SWIG[4] to generate Python wrappers for STAT's core functionality. The Pygments[5] Python module can optionally be installed to allow the STAT GUI to perform syntax highlighting of source code. Another GUI requirement is the Graphviz[6] package to render the DOT format graph files.

STAT can also be optionally built with the Fast Global File Status (FGFS)[7] 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/LLNL/MountPointAttributes and https://github.com/LLNL/FastGlobalFileStatus

Installation

STAT and its dependencies can also be built via the Spack package management tool, available at https://github.com/spack/spack. Running **spack install stat** should build STAT and all of its dependencies, including those require for the STAT GUI. Note that three Spack variants exist. The first is **+examples**, to enable building example MPI code that STAT can be tested against. This is disabled by default to avoid requiring an MPI library, but enabling it will trigger a build of MPI. The second is **+dysect** to enable building of the DySectAPI. The third is **+fgfs** to build with FastGlobalFileStatus and target binary broadcast support. FGFS is crucial to scalability, particularly when debugging applications that load several large dynamic libraries.

For STAT 4.0.1 and beyond, the spack build of STAT requires Python 3 and will require the activation of the py-xdot package **./bin/spack activate py-xdot**. For ver-

sions up to and including STAT 4.0.0, the spack build of STAT requires Python 2 and requires the activation of the py-pygtk and py-enum34 packages **./bin/spack activate py-pygtk** and **./bin/spack activate py-enum34**.

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 `--with-launchmon` and `--with-mrnet` configure options, thus it is important to specify both of these even if they share a prefix.

STAT will try to build the GUI by default. Building and running the STAT GUI requires SWIG and pygtk. If you need to modify your PYTHONPATH envirnment 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.

To compile on Cray systems runnig alps, you no longer need to specify `--with-cray-alps`. 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 \
MPICC=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.

STAT can also be configured to use GDB as a backend instead of Dyninst. To specify the path to gdb, use the `--with-gdb` flag. If CUDA kernal traces are desired,

the path should point to a working cuda-gdb executable. Note that STAT currently still requires Dyninst as a dependence even when using the GDB backend.

Notes

1. http://www.python.org/
2. http://www.pygtk.org/
3. https://pypi.org/project/xdot/
4. http://www.swig.org/
5. http://pygments.org/
6. http://www.graphviz.org/
7. https://github.com/LLNL/FastGlobalFileStatus

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-cl** 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`.

-z, --daemonspernode *num*

Sets the number of daemons per node to *num*.

-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].

-N, --nodesfile *filename*

Use the file *filename*, which should contain the list of nodes for communication processes

-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.

-P, --withpc

Sample program counter values in addition to function names.

-m, --withmoduleoffset

Sample module offset only.

-i, --withline

Sample source line number in addition to function names.

-o, --withopenmp

Translate OpenMP stacks to logical application view

-c, --comprehensive

Gather 5 traces: function only; module offset; function + PC; function + line; 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.

-H, --maxdaemonthreads *count*

Allow sampling of up to *count* threads per daemon.

-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 preferrably -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

[*FE* | *BE* | *CP*]

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.

-X, --dysectapi *session*

Run the specified DySectAPI *session*.

-b, --dysectapi_batch *secs*

Run the specified DySectAPI in batch mode. Session stops after *secs* seconds or detach action.

-G, --gdb

Use (cuda-)gdb to drive the daemons. If you are using cuda-gdb and want stack traces from cuda threads, you must also explicitly specify -w.

-Q, --cudaquick

When using cuda-gdb as the BE, gather less comprehensive, but faster cuda traces. Cuda frames will only show the top of the stack, not the full call path. This also defaults to display filename and line number and will not resolve the function name.

STAT Usage Example

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

```
% srun mpi_application arg1 arg2 &
[1] 16842

% 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 16842
```

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.

The **stat-view** GUI also allows you to view the application source files in the stack traces, when sampling is done at the source file and line number granularity. This may require the source file's path to be added to the search path, through **File -> Add Search Paths**. If an application's source code is edited after STAT is run, the line numbers shown in the stack traces may not be accurate. To alleviate this problem, STAT can optionally cache the source files for the currently displayed .dot file. To cache files click on the **File -> Add Search Paths** menu item. This will find and save the source files in the .dot file's stat_results directory. The next time you open the .dot file with **stat-view**, the source files will automatically be loaded from the cache.

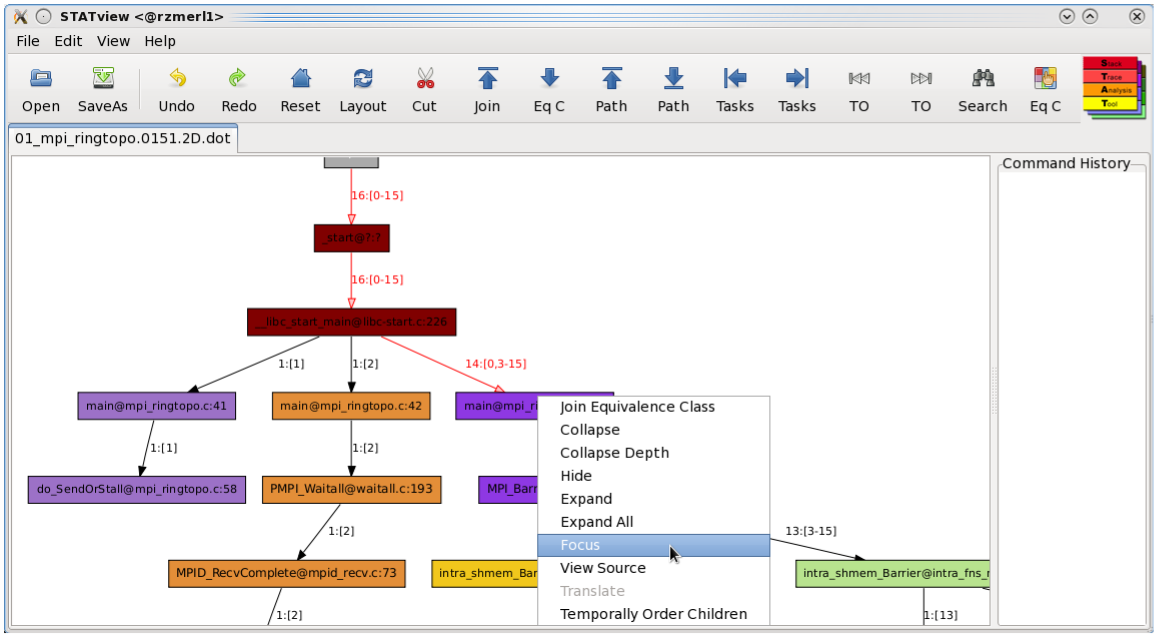


Figure 6-1. A screenshot of the stat-view GUI.

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.

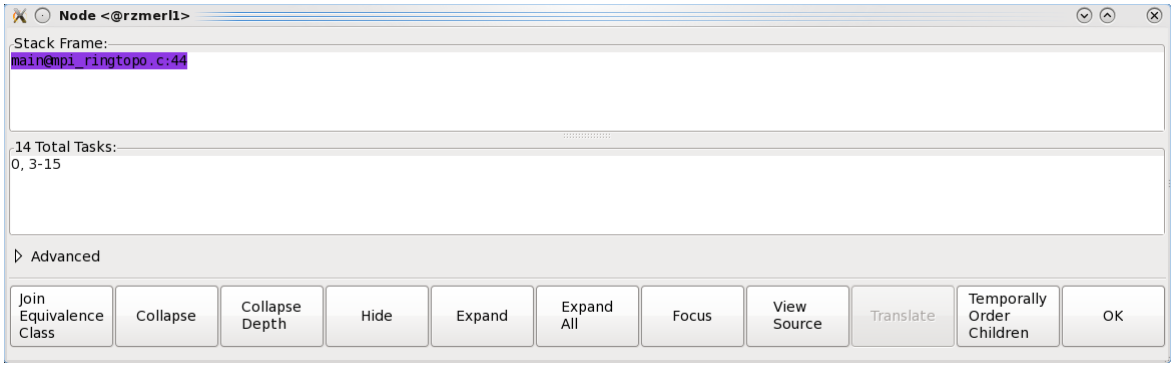


Figure 6-2. The node pop-up window

The node operations are defined as follows:

Advanced

Display the full node and edge attributes.

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**.

Translate

For traces with module and offset granularity, this button will translate all node labels into source file and line number information and open the resulting graph in a new tab.

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.

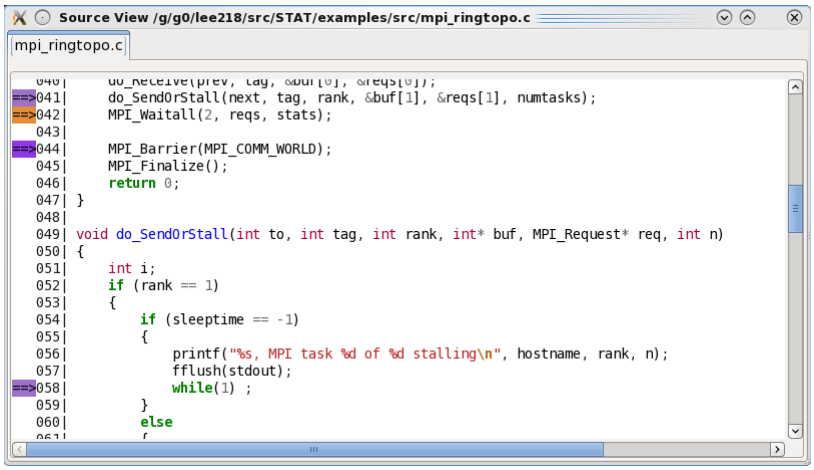


Figure 6-3. The source view window. The colored arrows correspond to the nodes in the call prefix tree.

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**.

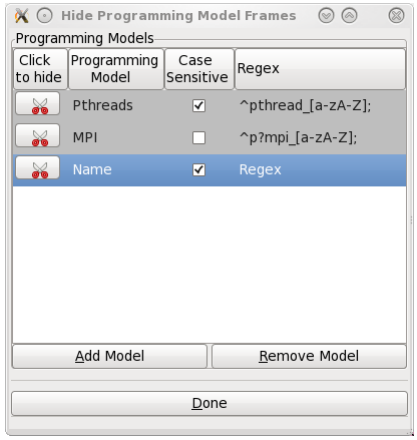


Figure 6-5. The stat-view programming model cutting interface.

[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.

[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.

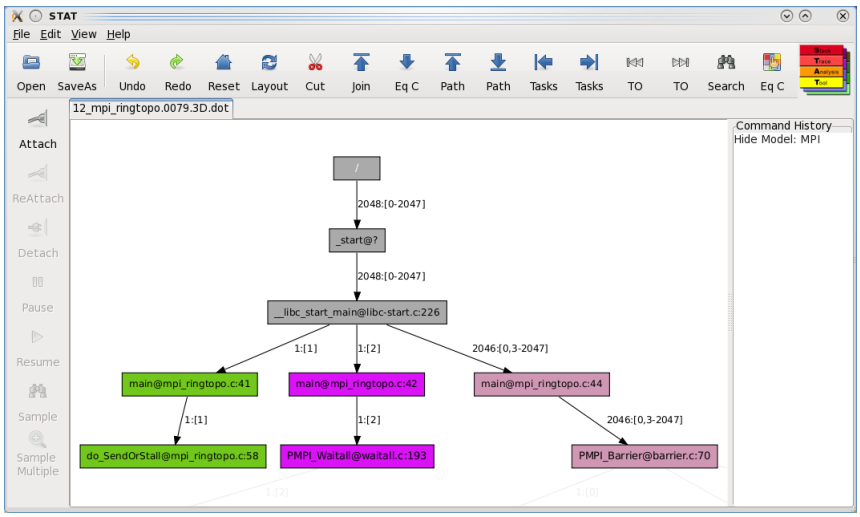


Figure 7-1. A screenshot of the STAT GUI

stat-gui Options

-a, --attach *[hostname:]PID*

Attach to the parallel job with resource manager *[hostname:]PID*.

-P, --withpc

Sample program counter values in addition to function names.

-m, --withmoduleoffset

Sample module offset only.

-i, --withline

Sample source line number in addition to function names.

-o, --withopenmp

Translate OpenMP stacks to logical application view

-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 preferrably -O0.

-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, --debugdaemons

launch the daemons under the deubgger

-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* | *CP*]

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.

-G, --gdb

Use (cuda-)gdb to drive the daemons. If you are using cuda-gdb and want stack traces from cuda threads, you must also explicitly specify -w.

-Q, --cudaquick

When using cuda-gdb as the BE, gather less comprehensive, but faster cuda traces. Cuda frames will only show the top of the stack, not the full call path. This also defaults to display filename and line number and will not resolve the function name.

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.

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).

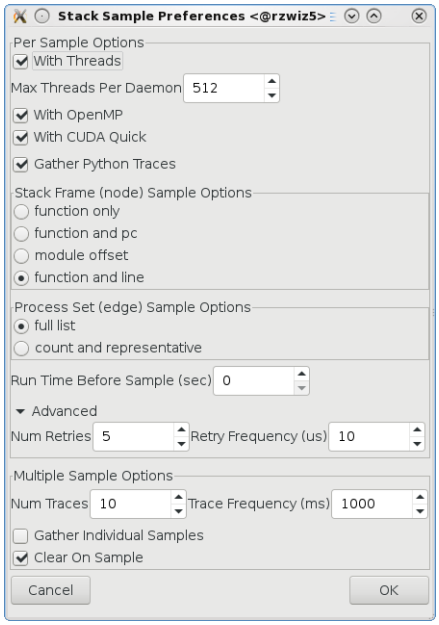


Figure 7-3. The stat-gui operation toolbar.

These options are defined as follows:

With Threads

Sample helper threads in addition to the main thread.

Max Threads Per Daemon*count*

Allow sampling of up to *count* threads per daemon.

With CUDA Quick

When using cuda-gdb as the BE, gather less comprehensive, but faster cuda traces. Cuda frames will only show the top of the stack, not the full call path. This also defaults to display filename and line number and will not resolve the function name.

With OpenMP

Translate OpenMP stacks into logical application view (requires application built with OMPD-enabled OpenMP)

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 preferrably -O0.

function only | module offset | function and pc | function and line

Sample traces with function name only, or module name and offset, or function name with the CPU program counter, or function name with the source file and line number. When gathering the module and offset, you can later translate all of the node labels into source file and line number via the GUI (left or right click on a node).

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.

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.

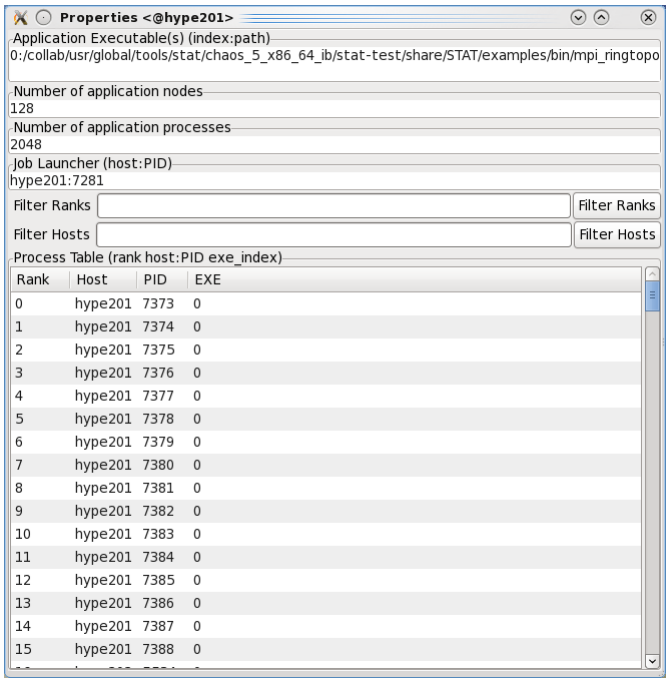


Figure 7-4. The properties window shows the application properties and lists the individual application processes.

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.

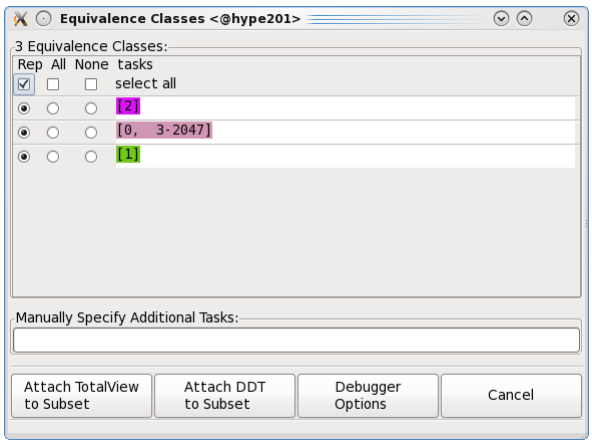


Figure 7-5. The equivalence classes window. The colored task lists correspond to the nodes in the prefix tree.

Availability

STAT has been ported to Linux x86 clusters, IBM BlueGene systems, CORAL and CORAL EA systems, Cray systems, and Intel Xeon Phi systems. It can be run against various resource managers, including SLURM, OpenMPI's OpenRTE, and Intel MPI's mpiexec.hyrdra.

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.

Resource Manager = *Auto|Alps|Slurm*

Sets the default resource manager to *Alps* or *Slurm* for searching for the job launcher process, or use *Auto* to determine the resourece manager automatically.

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.

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.

With OpenMP = *true|false*

Controls whether to translate OpenMP stack traces to logical application view.

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|module offset|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.

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_MRNET_OUTPUT_LEVEL enviornment variable).

GDB BE = *true|false*

Controls whether to use (cuda-)gdb to drive the deamons.

With CUDA QUICK = *true|false*

When using cuda-gdb as the BE, controls whether to gather less comprehensive, but faster cuda traces. Cuda frames will only show the top of the stack, not the full call path. This also defaults to display filename and line number and will not resolve the function name.

GDB Path = *path*

Use the gdb executable installed in *path* for debugging when GDB BE is set to true.

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_MRNET_OUTPUT_LEVEL=*level*

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

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_MRN_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_ADDR2LINE=*path*

Use the addr2line utility located in *path* to translate module and offset prefix trees in the stat-view GUI.

STAT_GDB=*path*

Use the gdb located in *path* to drive the daemons.

Chapter 9. Prescription-Based Debugging With Prototype DySectAPI

Overview

Debugging is a critical step in the development of any parallel program. However, the traditional interactive debugging model, where users manually step through code and inspect their application, does not scale well even for current supercomputers due its centralized nature. While lightweight debugging models, such as STAT, scale well, they can currently only debug a subset of bug classes. We therefore propose a new model, which we call prescriptive debugging, to fill this gap between these two approaches. This user-guided model allows programmers to express and test their debugging intuition in a way that helps to reduce the error space.

We have implemented a prototype implementation embodying the prescriptive debugging model, the DySectAPI, allowing programmers to construct probe trees for automatic, event-driven debugging at scale. The DySectAPI implementation can run with a low overhead.

The traditional debugging paradigm has survived because it provides the rudimentary operations that a user needs to effectively reduce the error search space. In a typical debug session, a user first sets a breakpoint at a particular code location. Once that breakpoint is triggered, the user will evaluate the state of the application and subsequently set another breakpoint, perhaps on a subset of processes that satisfy certain conditions. This process is then repeated until the bug is isolated.

Our new prescriptive debugging model aims to capture the flexibility and generality of this interactive process, but allow users to codify individual steps and sequences in the form of debug probes that can then be executed without the need for individual interactions between debugger and user. Essentially, the prescriptive debugging model provides the means for a user to codify their debugging intuition into prescribed debug sessions. The application can then be submitted into the system's batch queue to be run under that debug session.

At runtime, the debugger follows the user's intuition by executing the debug probes and, at the end, scalably gathers summary information that can be examined by the user during the execution or at their convenience after the job has completed. Our prescriptive parallel debugging model is built upon the notion of probes that can be linked together into a probe tree. A probe itself is composed of a *domain*, *events*, *conditions*, and *actions* as defined below.

The *domain* is the set of processes to install a probe into. It also includes a synchronization operation that determines how long the probe should wait for processes in the domain before proceeding. More precisely, after the first process triggers a probe, the remaining processes have until some specified timeout to participate.

We define an *event* as an occurrence of interest. Events borrowed from traditional debuggers include breakpoints, which specify a code location (when reached, the debugger will stop the target process) and data watchpoints, which monitor particular variables, memory locations or registers. An event can also be a user-defined timeout that instructs a probe to be triggered after some elapsed amount of time. Events can also capture asynchronous occurrences such as a program crash, a signal being raised or a system-level event such as memory exhaustion.

These events allow programmers to express their debugging in terms of a set of procedures and in terms of code behaviors (e.g., on detecting a hang or slowness). Further, individual events can also be composed together to enable advanced fine-grained event selection. When an event occurs, its associated *condition* is evaluated. The condition is an expression that can be evaluated either locally on each backend or globally across the domain. A local condition may, for instance, check if a variable equals a particular value. A global condition can evaluate an aggregated value, such as minimum, maximum or average, across the entire domain. Conditions can also be composed to specify multiple variables of interest or to combine local and global evaluations.

If the condition is satisfied, the probe is said to be *triggered*, and the specified *actions* are executed. Probe actions can be formulated by the user as an aggregation or a reduction, for example, aggregated messages, merged stack traces or the minimum and maximum of a variable. A probe can optionally include a set of child probes, which is enabled upon the satisfaction of the parent probe's condition. In this manner, a user can create a probe tree. A probe tree naturally matches the control-flow traversal that is typical of an interactive session with a traditional debugger. This can effectively narrow down the search space across the source-code dimension.

Installation

The DySectAPI comes included in STAT's source code. It can be built by turning on the `--enable-dysectapi` configure flag. DySectAPI requires DynInst library, which can be specified with the `--with-stackwalker=path` configure option.

Usage

A DySectAPI probe session is constructed as a C++ program. You will need to include the "DysectAPI.h" header file, which contains the specifications for the various DySectAPI objects and for the session routines. Your session must define a DysectStatus DysectAPI::onProcStart(int argc, char **argv) routine, which will create and link Probe objects. This section will only cover the high-level basics of constructing a debug session. For detailed information about the various constructs, you may refer to the header files or the reference guide.

A Probe consists of an Event, Condition, Domain, and an Action or list of Actions. Various Probe signatures exist, so not all components are required. Probes can be linked into a tree with the Probe::link(Probe *) routine. Probe tree roots can be enabled via the Probe::setup() routine.

An Event is an occurence of interest. DySectAPI defines several event types: Location; Timer; Async. Furthermore, Events can be combined with And, Or, and Not releations. A Location, essentially a debugger breakpoint, specifies where to install a probe. DySectAPI supports three location specifications: a function name; a source file and line number; or a program counter address. A Location can also be marked as pending, which informs DySectAPI that the probe should not be enabled until the parent probe has been triggered. A Timer specifies a timeout period to wait before triggering the Probe. An Async can be various asynchronous events, such as a signal, crash, or process exit.

A Condition specifies the circumstances under which a probe is to be triggered. Most conditions evaluate a data expression, such as whether a specified variable equals a certain value or falls within a given range. Conditions can be combined with And, Or, and Not relations.

The Domain specifies the set of processes within which to install a probe. The supported domains are World (i.e., all processes), a rank-specified Group, or Inherit (from the parent probe). A Domain can also specify a timeout, which indicates the amount of time to wait after the first process encounters the probe's event before proceeding.

An Action indicates what to do once a probe has been triggered. The currently implemented Actions are: trace, null, totalview, depositCore, signal, loadLibrary, irpc, writeModuleVariable, stat, detach, detachAll, stackTrace, fullStackTrace, startTrace, stopTrace. The trace action prints out an aggregated trace message. The totalview action detaches DySectAPI from the target processes that triggered the probe and attaches the TotalView debugger to that set of processes. The depositCore action, which requires the libdepositcore library, causes the triggering processes to dump a core file. The signal action sends the specified integer signal to the triggered processes. The loadLibrary action loads the specified shared library into the triggered processes. The irpc action will invoke a function within the target processes. the writeModuleVariable action will modify the contents of a variable. The stat action will gather a Stack Trace

Analysis Tool merged stack trace of the application. The detach action will detach DySectAPI from the triggered processes, while the detachAll action will detach DySectAPI from all application processes. The stackTrace and fullStackTrace actions will print an aggregated, text-based stack trace of the triggered processes. The startTrace and stopTrace actions indicate when to start and stop data tracing.

An example DySectAPI session can be seen below:

```
/* File: session.C */
#include <DysectAPI.h>
#include <stdio.h>

/* Single entry for debug daemon */
DysectStatus DysectAPI::onProcStart(int argc, char **argv) {

  /* Probe creation */
  Probe* entry  = new Probe(Code::location("entry(foo)"),
                            Local::eval("argc >= 5"),
                            Domain::group("12,25,65-70", Wait::inf));

  /* Within 500 ms and call frame has not been left */
  Probe* timer  = new Probe(Event::And(Time::within(500),
                            Event::Not(Async::leaveFrame())),
                            Domain::group("..", 400),
                            Action::trace("Took more than 500ms to return from
                                        foo()"));

  /* Event chain */
  entry->link(timer);

  /* Setup probe tree */
  entry->setup();

  return DysectOK;
}
```

The session can be compiled with the **dysectc** command.

```
% dysectc session.C
```

You will then need to run **stat-cl** and specify the generated session .so file with the **stat-cl** -X=path option, for example:

```
% stat-cl -X $PWD/libsession.so -C srun mpi_application
```

Chapter 10. Tips and Tricks Using STAT

Running STAT at scale

STAT is highly scalable and its default analysis has been shown to run effectively on jobs even over one million MPI tasks. Even so, at extreme scales, there are several options that may make STAT's operation even more scalable. The first is to specify the underlying communication tree topology. By default, the **stat-cl** command and **stat-gui** will try to deploy the automatic topology, which defaults to a fanout of 64. STAT will, by default, try to co-locate the communication processes on the application nodes (or associated I/O node on BG systems). To avoid the default co-location option for the **stat-cl** script (remove the `-A` option). For the **stat-gui** GUI, create a preferences file and specify an alternative `CP policy` option. Refer to the options sections to learn about more topology options.

Typically, STAT launches one debug daemon per node. This can become a bottleneck if that daemon is resposible for debugging many target processes. STAT has a `daemonspernode` that allows users to request that multiple daemons be launched per node and distribute the target processes between them. This will help offload some of the debugging workload, however, be aware that this will increase STAT's memory usage per node.

Another consideration at scale is the granularity of debug information. At larger scales, you may prefer to start with coarse-grained analysis. For example, you may not need full task lists for the edge labels, but rather, would like to gather edge labels with just the task count and a representative rank. Within the **stat-gui** GUI, one can then request the full task list of a given edge via the left-click menu of the desired edge's target node. Note, with the **stat-cl** command, the full task lists would not be gathered and thus would not be available for post-mortem analysis via the stat-view GUI. The count and representative granularity will result in faster sampling and smaller output-file size.

The granularity of the stack traces themselves can also be adjusted to alleviate bottlenecks at scale. In particular, symbol resolution can be expensive, particularly when gathering traces at the function and line level of granularity, but even with the function only granularity. This can cause the many STAT daemons to perform many file operation requests at the same time, straining the target file system. To alleviate this issue, you may gather stack traces with the module and offset granularity. The **stat-gui** and **stat-view** GUIs can later translate the module and offset into function and line number via **addr2line**. This translation feature is available only through a left-click of a module and offset node and will translate the entire prefix tree. Note that the file-system bottleneck can also be mitigated with the `--with-fgfs` configure option, to enable scalable file operations via the FastGlobalFileStatus[1] module.

Using STAT with IO Watchdog and SLURM

STAT can be used in conjunction with the IO Watchdog[2] 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

1. https://github.com/dongahn/FastGlobalFileStatus
2. http://code.google.com/p/io-watchdog/

Chapter 11. 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 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 11-1.

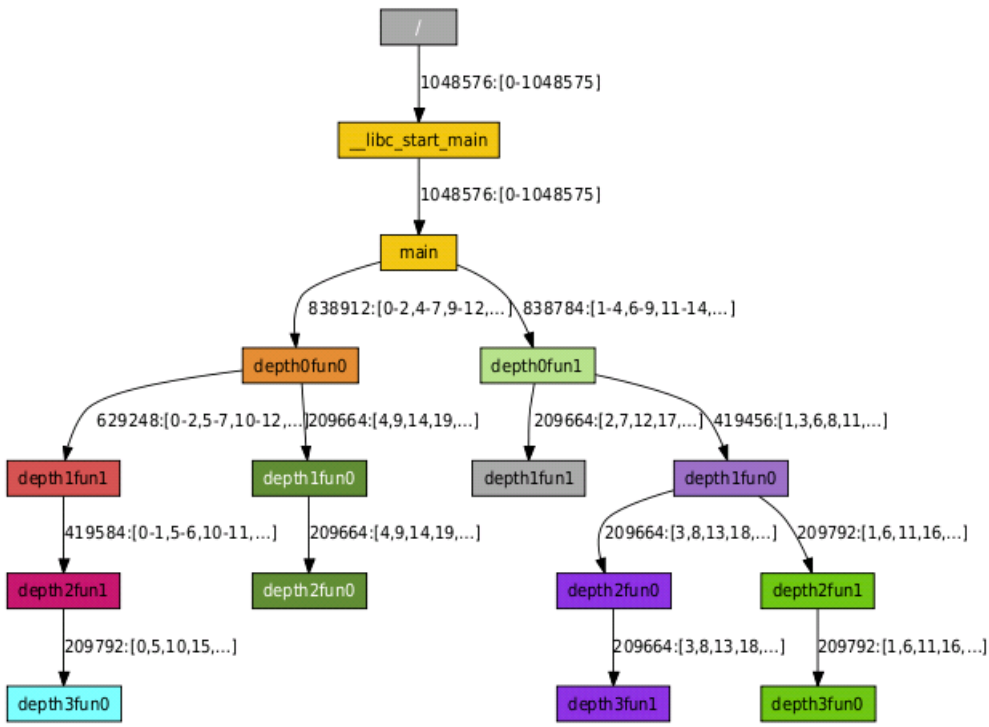


Figure 11-1. A stat-bench generated prefix tree emulating over 1 million tasks.

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).

-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 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 12. Using the stat-script Python Interface

Description

stat-script is a Python interface for STAT. When invoked without any arguments, this will create an interactive Python environment with the appropriate paths set to import the **STAT** module. You can run **dir(STAT)** to see the available functions, flags, etc. running **help()** (i.e., **help(STAT.attach)**) will display the description of the function as well as the arguments required. Like the **python** command, **stat-script** can also be supplied with a python script file to execute. You may refer to examples/scripts/script_test.py in the STAT source or share/STAT/examples/bin/script_test.py in the installation directory for an example of how the stat scripting interface can be used.

Chapter 13. Troubleshooting Guide

Troubleshooting

STAT hangs when attaching to Intel MPI jobs

When using the Intel MPI, you may need to alter your LaunchMON installation's etc/rm_intel_hydra.conf file and set RM_launch_helper=mpirun. If it is set to mpiexec.hydra, the daemons may fail to launch on remote nodes.

stack traces are empty

Some optimzations may make it impossible to debug, such as the GNU -fomit-frame-pointer option, which is enabled at various -O optimization levels. You can turn this off with the -fno-omit-frame-pointer flag.

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 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.

UnboundLocalError: local variable 'count' referenced before assignment

This error results from trying to open a STAT >3.0 output with a STAT 2.X GUI.

Hang on second attach when using CUDA GDB

In some CUDA environments, you may need to run your application with the CUDA_VISIBLE_DEVICES environment variable set appropriately, otherwise cuda-gdb will hang on attach when there is already an existing cuda-gdb session on the same node. Refer to the "Simultaneous Sessions Support" section of http://docs.nvidia.com/cuda/cuda-gdb/index.html.

Bibliography

Nicklas B. Jensen, Niklas Q. Nielsen, Gregory L. Lee, Dong H. Ahn, Sven Karlsson, Matthew P. Legendre, and Martin Schulz, "A Scalable Prescriptive Parallel Debugging Model," *International Parallel & Distributed Processing Symposium,* Hyderabad, India, May 2015.

Dong H. Ahn, Michael J. Brim, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Matthew P. Legendre, Barton P. Miller, Adam Moody, and Martin Schulz, "Efficient and Scalable Retrieval Techniques for Global File Properties," *International Parallel & Distributed Processing Symposium,* Boston, Massachusetts, May 2013.

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. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7161535
2. ftp://ftp.cs.wisc.edu/paradyn/papers/FGFS-IPDPS13-Ahn-validated.pdf
3. ftp://ftp.cs.wisc.edu/paradyn/papers/Miller09ScalableDebugging.pdf
4. ftp://ftp.cs.wisc.edu/paradyn/papers/Lee08ScalingSTAT.pdf
5. ftp://ftp.cs.wisc.edu/paradyn/papers/Ahn08LaunchMON.pdf
6. ftp://ftp.cs.wisc.edu/paradyn/papers/Lee07STATBench.pdf
7. ftp://ftp.cs.wisc.edu/paradyn/papers/Arnold06STAT.pdf