

## Open|SpeedShop Reference Guide

November 12, 2016

Version 2.3.0

Contributions from Krell Institute, LANL, LLNL, SNL

## Table of Contents

<b>About this Manual .....</b>	<b>6</b>
<b>1 Introduction to Open SpeedShop.....</b>	<b>8</b>
<b>1.1 Basic Concepts, Interface, Workflow.....</b>	<b>8</b>
1.1.1 Common Terminology .....	8
1.1.2 Concept of an Experiment .....	10
<b>1.2 Performance Experiments Overview .....</b>	<b>10</b>
1.2.1 Individual Experiment Descriptions .....	10
1.2.3 Overview of the Sampling Experiments .....	12
1.2.4 Overview of the Tracing Experiments.....	12
1.2.5 Parallel Experiment Support .....	13
<b>1.3 Running an O SS Experiment.....</b>	<b>13</b>
<b>1.4 How to Gather and Understand Profiles .....</b>	<b>19</b>
<b>2.1 Program Counter Sampling (pcsamp) Experiment .....</b>	<b>19</b>
2.1.1 Program Counter Sampling (pcsamp) experiment performance data gathering .	20
2.1.1.1 Program Counter Sampling (pcsamp) experiment parameters .....	20
2.1.2 Program Counter Sampling (pcsamp) experiment performance data viewing with GUI.....	21
2.1.3 Program Counter Sampling (pcsamp) experiment performance data viewing with CLI.....	21
<b>3.1 Call Path Profiling (usertime) Experiment.....</b>	<b>21</b>
3.1.1 Call Path Profiling (usertime) experiment performance data gathering .....	22
3.1.2 Call Path Profiling (usertime) experiment performance data viewing with GUI ..	23
3.1.3 Call Path Profiling (usertime) experiment performance data viewing with CLI ...	26
<b>4 How to Relate Data to Architectural Properties .....</b>	<b>30</b>
<b>4.1 Hardware Counter Experiment (hwc) .....</b>	<b>31</b>
4.1.1 Hardware Counter Threshold (hwc) experiment performance data gathering ....	32
4.1.2 Hardware Counter Threshold (hwc) experiment performance data viewing with GUI.....	32
4.1.3 Hardware Counter Threshold (hwc) experiment performance data viewing with CLI.....	34
<b>4.2 Hardware Counter Time Experiment (hwctime) .....</b>	<b>35</b>
4.2.1 Hardware Counter Time Threshold (hwctime) experiment performance data gathering.....	36
4.2.2 Hardware Counter Threshold (hwctime) experiment performance data viewing with GUI.....	37
4.2.3 Hardware Counter Time Threshold (hwctime) experiment performance data viewing with CLI.....	39
<b>4.3 Hardware Counter Sampling (hwcsamp) Experiment.....</b>	<b>42</b>
4.3.1 Hardware Counter Sampling (hwcsamp) experiment performance data gathering .....	44
4.3.1.1 Hardware Counter Sampling (hwcsamp) experiment parameters.....	44
4.3.2 Hardware Counter Sampling (hwcsamp) experiment performance data viewing with GUI.....	45
4.3.2.1 Getting the PAPI counter as the GUIs Source Annotation Metric.....	45

4.3.2.2 Viewing Hardware Counter Sampling Data with the GUI .....	47
4.3.3 Hardware Counter Sampling (hwcsamp) experiment CLI performance data viewing.....	48
4.3.3.1 Job Script and osshwcsamp command.....	48
4.3.3.2 osshwcsamp experiment default CLI view.....	49
4.3.3.2 osshwcsamp experiment Status command and CLI view .....	50
4.3.3.3 osshwcsamp experiment Load Balance command and CLI view.....	51
4.3.3.4 osshwcsamp experiment Linked Object command and CLI view.....	51
4.3.3.5 osshwcsamp experiment only the hwcsamp PAPI events CLI view .....	51
<b>5 I/O Tracing and I/O Profiling .....</b>	<b>52</b>
<b>5.1 Open SpeedShop I/O Tracing General Usage .....</b>	<b>52</b>
<b>5.2 I/O Base Tracing (io) experiment .....</b>	<b>52</b>
5.2.1 I/O Base Tracing (io) experiment performance data gathering.....	52
5.2.2 I/O Base Tracing (io) experiment performance data viewing with CLI.....	52
5.2.3 I/O Base Tracing (io) experiment performance data viewing with GUI.....	52
<b>5.3 I/O Extended Tracing (iot) experiment .....</b>	<b>53</b>
5.3.1 I/O Extended Tracing (iot) experiment performance data gathering .....	53
5.3.2 I/O Extended Tracing (iot) experiment performance data viewing with GUI.....	53
5.3.3 I/O Extended Tracing (iot) experiment performance data viewing with CLI.....	56
<b>5.3 I/O Lightweight Profiling (iop) General Usage .....</b>	<b>58</b>
5.3.1 I/O Profiling (iop) experiment performance data gathering .....	58
5.3.2 I/O Profiling (iop) experiment performance data viewing with GUI .....	59
5.3.3 I/O Profiling (iop) experiment performance data viewing with CLI.....	60
<b>6 Applying Experiments to Parallel Codes.....</b>	<b>63</b>
<b>7 MPI Tracing Experiments (mpi, mpit, mpip) .....</b>	<b>65</b>
<b>7.1 MPI Tracing Experiment (mpi) .....</b>	<b>75</b>
7.1.1 MPI Tracing Experiment (mpi) performance data gathering .....	75
7.1.2 MPI Tracing Experiment (mpi) performance data viewing with GUI.....	76
7.1.3 MPI Tracing Experiment (mpi) performance data viewing with CLI.....	76
<b>7.2 MPI Tracing Experiments (mpit).....</b>	<b>78</b>
7.2.1 MPI Tracing Experiments (mpit) performance data gathering .....	78
7.2.2 MPI Tracing Experiments (mpit) performance data viewing with GUI .....	78
7.2.3 MPI Tracing Experiments (mpit) performance data viewing with CLI.....	78
<b>7.3 MPI Tracing Experiments (mpip).....</b>	<b>80</b>
7.3.1 MPI Tracing Experiments (mpip) performance data gathering .....	80
7.3.3 MPI Tracing Experiments (mpip) performance data viewing with GUI .....	83
<b>8 Threading Analysis Section .....</b>	<b>85</b>
<b>8.1 Threading Specific Experiment (pthreads).....</b>	<b>87</b>
8.1.1 Threading Specific (pthreads) experiment performance data gathering.....	87
8.1.2 Threading Specific (pthreads) experiment performance data viewing with GUI.....	87
8.1.3 Threading Specific (pthreads) experiment performance data viewing with CLI.....	89
<b>8.2 OpenMP Related Performance Analysis.....</b>	<b>90</b>
<b>8.2.1 OpenMP Thread Wait Detection using OMPT interface.....</b>	<b>90</b>
<b>8.2.1.1 Augmentation of Open SpeedShop sampling experiments.....</b>	<b>90</b>
<b>8.2.2 Open SpeedShop OpenMP specific profiling experiment (omptp) .....</b>	<b>95</b>
<b>8.3 Hybrid (openMP and MPI) Performance Analysis .....</b>	<b>95</b>
<b>8.3.1 Focus on individual Rank to get Load Balance for Underlying Threads.....</b>	<b>96</b>

8.3.2 Clearing Focus on individual Rank to get bank to default behavior .....	98
<b>9 GPU Performance Analysis .....</b>	<b>101</b>
<b>9.1 NVIDIA CUDA Analysis Section.....</b>	<b>101</b>
9.1.1 NVIDIA CUDA Tracing (cuda) experiment performance data gathering.....	101
9.1.2 NVIDIA CUDA Tracing (cuda) experiment performance data viewing with GUI	102
9.1.3 NVIDIA CUDA Tracing (cuda) experiment performance data viewing with CLI	103
<b>10 Memory Analysis Techniques.....</b>	<b>109</b>
<b>10.1 Memory Analysis Tracing (mem) experiment performance data gathering .</b>	<b>109</b>
<b>10.2 Memory Analysis Tracing (mem) experiment performance data viewing with</b>	
<b>CLI.....</b>	<b>109</b>
10.2.1 Additional CLI Information .....	111
<b>10.3 Memory Analysis Tracing (mem) experiment performance data viewing with</b>	
<b>GUI .....</b>	<b>114</b>
<b>11 Advanced Analysis Techniques.....</b>	<b>117</b>
<b>11.1 Comparison Script Argument Description.....</b>	<b>118</b>
11.1.1 osscompare metric argument.....	118
11.1.2 osscompare rows of output argument .....	119
11.1.3 osscompare output name argument. ....	119
11.1.4 osscompare view type or granularity argument. ....	120
<b>12 Open SpeedShop User Interfaces .....</b>	<b>120</b>
<b>12.1 Command Line Interface Basics .....</b>	<b>121</b>
12.1.2 CLI Metric Expressions and Derived Types .....	122
12.1.3 CLI Automatically Generated Derived Metrics and CLI Derived Metric Names	124
12.1.3.1 Computational Intensity.....	124
12.1.3.2 Level 1 Data Cache Miss Ratio .....	124
<b>12.2 CLI Batch Scripting .....</b>	<b>124</b>
<b>12.3 Python Scripting .....</b>	<b>125</b>
<b>12.4 MPI_Pcontrol Support .....</b>	<b>125</b>
<b>12.5 Graphical User Interface Basics .....</b>	<b>126</b>
12.5.1 Basic Initial View – Default View .....	126
12.5.1.1 Icon ToolBar.....	126
12.5.1.2 View/Display Choice Selection.....	127
12.5.2 Preferences - How to change preferences .....	129
12.5.2.1 Disabling or enabling the preference for Save/Reuse views in CLI. ....	131
<b>13 Special System Support (Static Executables).....</b>	<b>133</b>
<b>13.1 Cray and Blue Gene .....</b>	<b>133</b>
<b>13.1.1 osslink Command Information .....</b>	<b>134</b>
<b>13.1.2 Cray Specific Static aprun Information.....</b>	<b>135</b>
<b>13.1.3 Changing parameters to the experiments .....</b>	<b>135</b>
<b>14 Setup and Build for Open SpeedShop .....</b>	<b>137</b>
<b>14.1 Open SpeedShop Cluster Install.....</b>	<b>137</b>
<b>14.2 Open SpeedShop Blue Gene Platform Install .....</b>	<b>138</b>
<b>14.3 Open SpeedShop Cray Platform Install.....</b>	<b>138</b>
<b>14.4 Execution Runtime Environment Setup .....</b>	<b>138</b>
14.4.1 Example module file .....	138
14.4.2 Example softenv file .....	139
14.4.3 Example dotkit file .....	140

<b>15 Additional Information and Documentation Sources .....</b>	<b>140</b>
15.1 Final Experiment Overview .....	140
15.2 Additional Documentation .....	141
<b>16 Convenience Script Basic Usage Reference Information .....</b>	<b>142</b>
16.1 Suggested Workflow .....	142
16.2 Convenience Scripts .....	142
16.3 Report and Database Creation .....	142
16.4 osscompare: Compare Database Files .....	143
16.5 osspcsamp: Program Counter Experiment .....	143
16.6 ossusertime: Call Path Experiment .....	143
16.7 osshwc, osshwctime: HWC Experiments .....	144
16.8 osshwcsamp: HWC Experiment .....	144
16.9 ossio, ossiot: I/O Experiments .....	144
16.10 ossmpi, ossmpip, ossmpit: MPI Experiments .....	145
16.11 ossmem: Memory Analysis Experiment .....	145
16.12 ossomptp: OpenMP Specific Profiling Experiment .....	146
16.13 osspthreads: POSIX Thread Analysis Experiment .....	146
16.14 osscuda: NVIDIA CUDA Tracing Experiment .....	146
16.15 Key Environment Variables .....	146

## About this Manual

This reference guide provides basic Open|SpeedShop (O|SS) usage information.

This manual intends to give users an understanding of the general experiments available in O|SS that can be used to analyze application code. There is extensive information provided about how to use the O|SS experiments and how to view the performance information in informative ways. Hopefully this will allow users to start optimizing and analyzing the performance of application code.

O|SS is a community effort by [The Krell Institute](#) with current direct funding from the Department of Energy's National Nuclear Security Administration (DOE NNSA). It builds on a broad list of community provided infrastructures, notably the Paradyn Project's Dyninst API and MRNet (Multicast Reduction Network) from the University of Wisconsin at Madison, the Libmonitor profiling tool, and the Performance Application Programming Interface (PAPI) from the University of Tennessee at Knoxville. O|SS is an open source multi-platform Linux performance tool which is targeted to support performance analysis of applications running on both single node and large scale Intel, AMD, ARM, Intel Phi, PPC, GPU processor based systems and on Blue Gene and Cray platforms.

O|SS is explicitly designed with usability in mind and is for application developers and computer scientists. The base functionality includes:

- Overview Program Counter Sampling Experiment
- Support for Call Stack Analysis
- Access to the Hardware Performance Counters
- MPI Profiling and Tracing
- I/O Profiling and Tracing
- Memory Analysis Tracing
- POSIX Thread Function Tracing
- NVIDIA CUDA Event Tracing
- OpenMP Profiling and Analysis

In addition, O|SS is designed to be modular and extensible. It supports several levels of plug-ins, which allow users to add their own performance experiments.

O|SS development is hosted by the Krell Institute. The infrastructure and base components of O|SS are released as open source code primarily under LGPL.

Highlights include:

- Comprehensive performance analysis for sequential, multithreaded, and MPI applications
- No need to recompile the user's application.
- Supports both first analysis steps as well as deeper analysis options for performance experts

- Easy to use GUI and fully scriptable through a command line interface and Python
- Supports Linux Systems and Clusters with Intel and AMD processors
- Extensible through new performance analysis plugins ensuring consistent look and feel
- In production use on all major cluster platforms at LANL, LLNL, and SNL and other sites around the world

Features include:

- Four user interface options: batch, command line interface, graphical user interface and Python scripting API.
- Supports multi-platform single system image (SSI) and traditional clusters.
- Scales to large numbers of processes, threads, and ranks.
- View performance data using multiple customizable views.
- Save and restore performance experiment data and symbol information for post experiment performance analysis.
- View performance data for all of application's lifetime or smaller time slices.
- Compare performance results between processes, threads, or ranks between a previous experiment and current experiment.
- Interactive CLI help facility, which lists the CLI commands, syntax, and typical usage.
- Option to automatically group like-performing processes, threads, or ranks.

# 1 Introduction to Open|SpeedShop

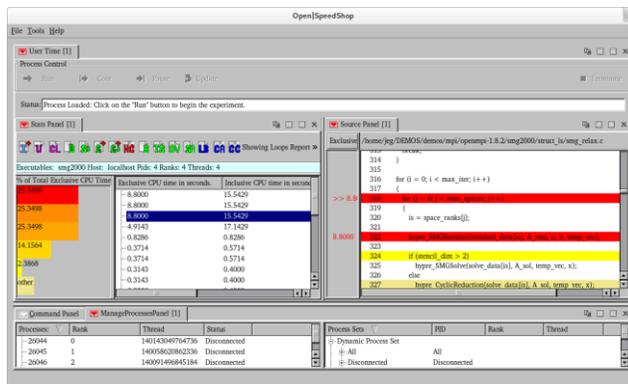
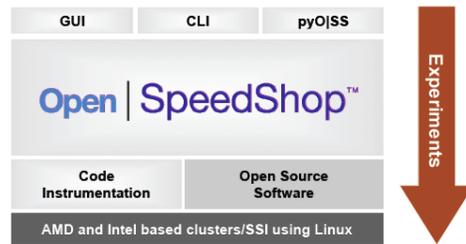
Open|SpeedShop (O|SS) is an open source performance analysis tool framework. It provides the most common performance analysis steps all in one tool using a common shared interface. It is easily extendable by writing plugins to collect and display performance data. It also comes with built in experiments to gather and display several types of performance information.

O|SS provides several flexible and easy ways to interact with it. There is a GUI to launch and examine experiments, a command line interface that provides the same access as the GUI, as well as python scripts. There are also convenience scripts that allow you to run standalone experiments on applications and examine the results at a later time.

The existing experiments for O|SS all work on unmodified application binaries. O|SS has been tested on a variety of Linux clusters and supports Cray and Blue Gene systems.

## 1.1 Basic Concepts, Interface, Workflow

O|SS has three ways for the user to examine the results of a performance test, called experiments, a GUI, a command line interface or through python libraries. The user can also start experiments by using those three options or by an additional method of the command line launched convenience scripts. For example, to launch one of the convenience scripts for the pcsamp experiment (Program Counter Sampling) the user executes the command `osspcsamp <application>`, where `<application>` is the executable under study along with any arguments. The convenience scripts will then create a database for the results of that experiment.



The user can examine any database in the GUI with the command:

```
openss -f <db file>
```

The GUI will provide some simple graphics to help you understand the results and will relate the data back to the source code when possible.

### 1.1.1 Common Terminology

Technical terms can have multiple and/or context sensitive meanings, therefore this section attempts to explain and clarify the meanings of the terms used in this document, especially with respect to the O|SS tools.

- Experiment:** A set of collectors and executable or executables bound together to generate performance information that can be viewed in human readable form.
- Focused Experiment:** The current experiment commands operate on. The user may run or view multiple experiments simultaneously and unless a particular experiment is specified directly, the focused experiment will be used. Experiments are given an enumeration, called an experiment id, for identification.
- Component(s):** A component is a somewhat self-contained code section of the O|SS performance tool. This section of code does a set of specific related tasks for the tool. For example, the GUI component does all the tasks related to displaying O|SS wizards, experiment creation, and results using a graphical user interface. The CLI component does similar functions but uses the interactive command line delivery method.
- Collector:** The portion of the tool containing logic that is responsible for the gathering of the performance metric. A collector is a portion of the code that is included in the experiment plugin.
- Metric:** The measurement, which the collector/experiment is gathering. A metric could be a time, an occurrence counter, or other entity, which reflects in some way on the application's performance and is gathered by a performance experiment at application runtime directly by the collector.
- Offline:** offline is the O|SS default mode of operation. It is a link override mechanism that allows for gathering performance data using libmonitor to link O|SS performance data gathering software components into the user application. For this mode of operation, the application must be run from start up to completion. The performance results may be viewed after the application terminates normally.
- Param:** Each collector allows the user to set certain values that control the way a collector behaves. The parameter or **param** may cause the collector to perform various operations at certain time intervals or it may cause a collector to measure certain types of data. Although O|SS provides a standard way to set a parameter, it is up to the individual collector to decide what to do with that information. Detailed documentation about the available parameters is part of the collector's documentation.
- Framework:** The set of API functions that allows the user interface to manage the creation and viewing of performance experiments. It is the interface between the user interface and the cluster support and dynamic instrumentation components.
- Plugin:** A portion (library) of the performance tool that can be loaded and

included in the tool at tool start-up time. Development of the plugin uses a tool specific interface (API) so that the plugin, and the tool it is to be included in, know how to interact with each other. Plugins are normally placed in a specific directory so that the tool knows where to find the plugins.

**Target:** This is the application or part of the application that O|SS is running the experiment on. In order to fine tune what is being targeted, O|SS gives target options that describes file names, host names, thread identifiers, rank identifiers and process identifiers.

### **1.1.2 Concept of an Experiment**

O|SS uses the concept of an experiment to describe the gathering of performance measurement data for a particular performance area of interest. Experiments consist of the performance data collector responsible for the gathering of the measurements associated with the performance area of interest. The collector, which is a small dynamic or static object library, also contains functions that can interpret the gathered measurements, i.e., performance data, into a human understandable form. The experiment definition also includes the application being examined and how often the data will be gathered (the sampling rate). The application's symbol information is saved into the experiment output file so that performance reports can be generated from the performance data file alone. The application, itself, need not be present to view the performance data at a later time.

## **1.2 Performance Experiments Overview**

O|SS refers to the different performance measurements as experiments. Each experiment can measure and analyze different aspects of the code's performance. The experiment type, or type of data gathered, is chosen by the user. Any experiment can be applied to any application, with the exception of MPI specific experiments being applied to non-MPI applications.

Each experiment consists of collectors and views. The collectors define specific performance data sources, for example, program counter samples, call stack samples, hardware counters or tracing of library routines. Views specify how the performance data is aggregated and presented to the user. It is possible to implement multiple collectors per experiment.

### **1.2.1 Individual Experiment Descriptions**

The following table provides a quick overview of the different experiment types that come with O|SS.

<b>Experiment</b>	<b>Experiment Description</b>
<b>pcsamp</b>	Periodic sampling the program counters gives a low-overhead view of where the time is being spent in the user application.
<b>usertime</b>	Periodic sampling the call path allows the user to view inclusive and exclusive time spent in application routines. It also allows the user to see which routines called which routines. Several views are available, including the “hot” path.
<b>hwc</b>	Hardware events (including clock cycles, graduated instructions, instruction and data cache and TLB misses, floating-point operations) are counted at the machine instruction, source line and function levels.
<b>hwcsamp</b>	Similar to hwc, except that sampling is based on time, not PAPI event overflows. Up to six events may be sampled during the same experiment.
<b>hwctime</b>	Similar to hwc, except that call path sampling is also included <small>[L] [SEP]</small>
<b>io</b>	Accumulated wall-clock durations of input/output (I/O) system calls: read, readv, write, writev, open, close, dup, pipe, creat and others. Show call paths for each unique I/O call path.
<b>iop</b>	Lightweight I/O profiling: Accumulated wall-clock durations of I/O system calls: read, readv, write, writev, open, close, dup, pipe, creat and others, but individual call information is not recorded.
<b>iot</b>	Similar to io, except that more information is gathered, such as bytes moved, file names, etc.
<b>mpi</b>	Captures the time spent in and the number of times each MPI function is called. Show call paths for each MPI unique call path.
<b>mpip</b>	Lightweight MPI profiling: Captures the time spent in and the number of times each MPI function is called. Show call paths for each MPI unique call path, but individual call information is not recorded.
<b>mpit</b>	Records each MPI function call event with specific data for display using a GUI or a command line interface (CLI). Trace format option displays the data for each call, showing its start and end times.
<b>mem</b>	Tracks potential memory allocation call that is not later destroyed (leak). Records any memory allocation event that set a new high-water of allocated memory current thread or process. Creates an event for each unique call path to a traced memory call and records the total number of times this call path was followed, the max allocation size, the min allocation size, and the total allocation, the total time spent in the call path, and the start time for the first call.
<b>pthread</b>	Captures the time spent in and the number of times each POSIX thread function is called. Show call paths for each POSIX thread function’s unique call path
<b>omptp</b>	Report task idle, barrier, and barrier wait times per OpenMP thread and attribute those times to the OpenMP parallel regions.

<b>cuda*</b>	Captures the NVIDIA CUDA events that occur during the application execution and report times spent for each event, along with the arguments for each event, in an event-by-event trace.
--------------	---

\* Not available in OJSS using the offline mode of operation, at this time.

### 1.2.3 Overview of the Sampling Experiments

Program counter sampling (pcsamp) experiment, call path profiling (usertime) experiment, and the three hardware counter experiments (hwc, hwctime, hwcsamp) all use a form of sampling based performance information gathering techniques.

Program Counter Sampling (pcsamp) is used to record the Program Counter (PC) in the user application being monitored by interrupting the application at a user defined time interval, with the default being 100 times a second. This experiment provides a low overhead overview of the time distribution for the application. Its lightweight overview provides a good first step for analyzing the performance of an application.

The Call Path Profiling (usertime experiment) gathers both the PC sampling information and also records call stacks for each sample. This allows the later display of the call path information about the application as well as inclusive and exclusive timing data (see section 4.2). This experiment is used to find hot call paths (call paths that take the most time) and see who is calling whom.

The Hardware Counter experiments (hwc, hwctime, hwcsamp) access data like Cache and TLB misses. The experiments hwc and hwctime, sample a hardware counter events, based on an event threshold. The default event is PAPI\_TOT\_CYC overflows. Please see chapter 5 for more information on PAPI and hardware counter related experiments. Instead using a threshold, the hwcsamp experiment samples up to six events based on a sample time, similar to the usertime and pcsamp experiments. The hwcsamp experiment default events are PAPI\_FP\_OPS and PAPI\_TOT\_CYC.

### 1.2.4 Overview of the Tracing Experiments

The Input/Output tracing and profiling experiments (io, iot, iop), MPI Tracing Experiments (mpi, mpip, mpit), Memory tracing (mem), and the POSIX thread tracing (pthread) all use a form of tracing or wrapping of the function names to record performance information. Tracing experiments do not use timers or thresholds to interrupt the application. Instead they intercept the function calls of interest by using a wrapper function that records timing and function argument information, calls the original function, and then records this information for later viewing with OJSS's user interface tools.

The Input/Output tracing experiments (io, iot) record invocation of all POSIX I/O events. They both provide aggregated and individual timings and, in addition, the iot experiment also provides argument information for each call. To obtain a more lightweight overview of application I/O usage, use the I/O profiling experiment. The lightweight I/O experiment (iop) records the invocation of all POSIX I/O events, accumulating the information, but does not save individual call information like the io and iot experiments do. That allows the iop experiment database to be smaller and makes the iop experiment faster than the io and iot experiments.

The memory tracing experiment (mem) records invocation of all tracked memory function calls, also referred to as events. The mem experiment provides aggregated and individual timings and also provides argument information for each call.

The MPI Tracing Experiments (mpi, mpit) record invocation of all MPI routines as well as aggregated and individual timings. The mpit experiment provides argument information for each call. To obtain a more lightweight overview of application MPI usage, use the MPI profiling experiment (mpip). The lightweight MPI experiment (mpip) records the invocation of all MPI function call events, accumulating the information, but does not save individual call information like the mpi and mpit experiments do. That allows the mpip experiment database to be smaller and makes the mpip experiment faster than the mpi and mpit experiments.

The POSIX thread tracing experiment (pthreads) records invocation of all tracked POSIX thread related function calls, also referred to as events. The pthreads experiment provides aggregated and individual timings and also provides argument information for each call.

### **1.2.5 Parallel Experiment Support**

O|SS supports MPI and threaded codes; it has been tested with a variety of MPI implementations. The thread support is based on POSIX threads and OpenMP is supported in a number of ways including via POSIX threads, OpenMP wait time augmentation of the sampling experiment and through the omptp OpenMP profiling experiment.

Any O|SS experiment can be applied to any parallel application. This means you can run the program counter sampling experiment on a non-parallel application as well as a MPI or threaded application. The experiment data collectors are automatically applied to all tasks/threads. The default views aggregate (sum the performance data) across all tasks/threads but data from individual tasks/threads are available. The MPI calls are wrapped, and MPI function elapsed time and parameter information is displayed.

## **1.3 Running an O|SS Experiment**

First think about what parameters you want to measure then choose the appropriate experiment to run. You may want to start by running the pcsamp experiment since it is a lightweight experiment and will give an overview of the timing for the entire application.

Once you have selected the experiment to run you can launch it with either the wizard in the GUI or by using the command line convenience scripts. For example, say you have decided to run the pcsamp experiment on the Semi coarsening Multigrid Solver MPI application smg2000. On the command line you would issue the command:

```
> osspcsamp "mpirun -np 256 smg2000 -n 60 60 60"
```

Where "mpirun -np 256 smg2000 -n 60 60 60" is a typical MPI application launching command you would normally use to launch the smg2000 application. mpirun, a MPI driver script or executable, is used to launch smg2000 on 256 processors with "-n 60 60 60" is passed as an argument to smg2000.

An example of a MPI smg2000 pcsamp experiment run from a SLURM based system using "srun" as the MPI driver, along with the application and experiment output follows below:

```
> osspcsamp "srun -n 256 ./smg2000 -n 60 60 60"
```

```
[openss]: pcsamp experiment using the pcsamp experiment default sampling rate: "100".  
[openss]: pcsamp experiment calling openss.  
[openss]: Setting up offline raw data directory in /p/lscratchrb/jeg/offline-oss  
[openss]: Running offline pcsamp experiment using the command:  
"srun -ppdebug -n 256 /collab/usr/global/tools/openspeedshop/oss-  
dev/x8664/oss_offline_v2.1u6/bin/ossrun -c pcsamp ./smg2000 -n 60 60 60"
```

```
Running with these driver parameters:  
(nx, ny, nz) = (60, 60, 60)  
(Px, Py, Pz) = (256, 1, 1)  
(bx, by, bz) = (1, 1, 1)  
(cx, cy, cz) = (1.000000, 1.000000, 1.000000)  
(n_pre, n_post) = (1, 1)  
dim = 3  
solver ID = 0
```

```
=====  
Struct Interface:  
=====
```

```
Struct Interface:  
wall clock time = 0.020830 seconds  
cpu clock time = 0.030000 seconds  
=====
```

```
Setup phase times:  
=====
```

```
SMG Setup:  
wall clock time = 0.451188 seconds  
cpu clock time = 0.460000 seconds  
=====
```

```
Solve phase times:
```

```

=====
SMG Solve:
wall clock time = 2.707334 seconds
cpu clock time = 2.720000 seconds
Iterations = 7
Final Relative Residual Norm = 1.446921e-07
[openss]: Converting raw data from /p/lscratchrb/jeg/offline-oss into temp file X.0.openss
Processing raw data for smg2000 ...
Processing processes and threads ...
Processing performance data ...
Processing symbols ...
Resolving symbols for /g/g24/jeg/demos/workshop_demos/mpi/smg2000/test/smg2000
Resolving symbols for /lib64/ld-2.12.so
Resolving symbols for /collab/usr/global/tools/openspeedshop/oss-
dev/x8664/oss_offline_v2.1u6/lib64/openspeedshop/pcsamp-rt-offline.so
Resolving symbols for /collab/usr/global/tools/openspeedshop/oss-
dev/x8664/krellroot_v2.1u6/lib64/libmonitor.so.0.0.0
Resolving symbols for /usr/local/tools/mvapich-gnu-1.2/lib/shared/libmpich.so.1.0
Resolving symbols for /lib64/libc-2.12.so
Resolving symbols for /lib64/libpthread-2.12.so
Resolving symbols for /usr/lib64/libpsm_infinipath.so.1.14
Resolving symbols for /usr/lib64/libinfinipath.so.4.0
Updating database with symbols ...
Finished ...

```

```

[openss]: Restoring and displaying default view for:
/g/g24/jeg/demos/workshop_demos/mpi/smg2000/test/smg2000-pcsamp.openss
[openss]: The restored experiment identifier is: -x 1

```

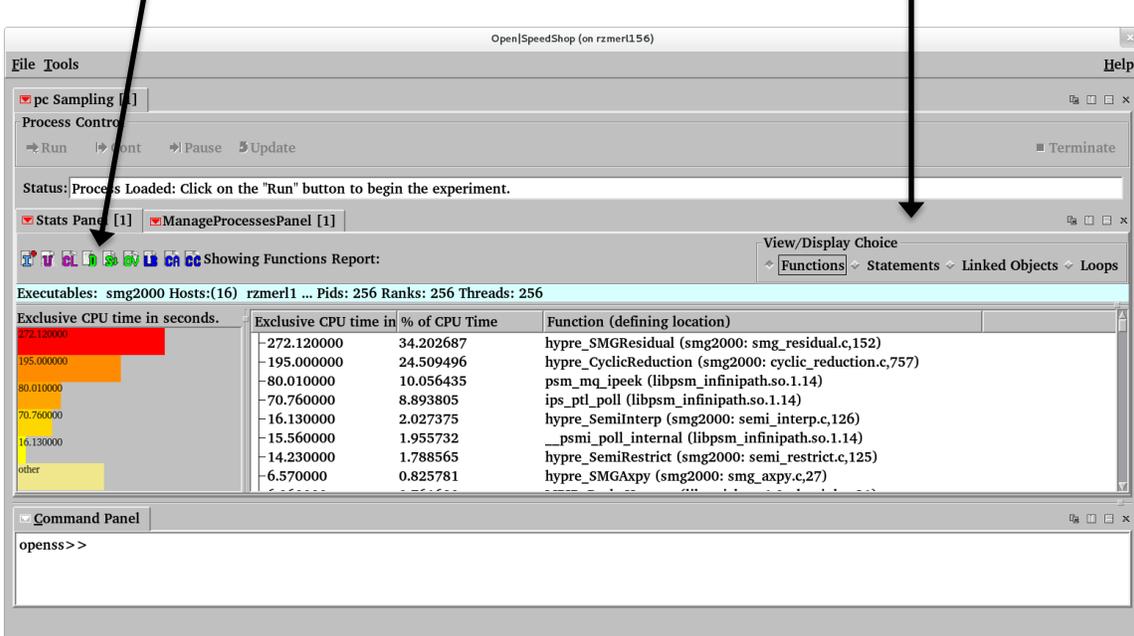
Exclusive CPU time	% of CPU Time	Function (defining location)
		in seconds.
272.1200	34.202	hypr_SMGResidual (smg2000: smg_residual.c,152)
195.0000	24.509	hypr_CyclicReduction (smg2000: cyclic_reduction.c,757)
80.0100	10.056	psm_mq_ipeek (libpsm_infinipath.so.1.14)
70.7600	8.893	ips_ptl_poll (libpsm_infinipath.so.1.14)
16.1300	2.027	hypr_SemiInterp (smg2000: semi_interp.c,126)
15.5600	1.955	_psmi_poll_internal (libpsm_infinipath.so.1.14)
14.2300	1.788	hypr_SemiRestrict (smg2000: semi_restrict.c,125)
6.5700	0.825	hypr_SMGAxy (smg2000: smg_axpy.c,27)
6.0600	0.761	MPIR_Pack_Hvector (libmpich.so.1.0: dmpipk.c,31)
5.9500	0.747	ipath_dwordcpy (libinfinipath.so.4.0)
5.7900	0.727	MPID_DeviceCheck (libmpich.so.1.0: psmcheck.c,35)
...		
...		

When the application completes a default report will be printed on screen. The performance information gathered during execution of the experiment will be stored in a database called smg2000-pcsamp.openss. You can use the OJSS GUI to analyze the data in detail. Run the openss command to load that database file or open the file directly using the “-f” option: `> openss -f smg2000-pcsamp.openss`

Below we show basic examples of how to use the GUI to view the output database file created by the convenience script.

View Type Choices (D for Default selected and shown)  
 LB: Load Balance, CA: Cluster Analysis, and others available.

View Granularity Choices (Function level selected)  
 Statement, Library, and Loop level are available.



You can choose to view data for the Function, Statement, Linked Object, or Loop level granularity. To switch from one view type to another, first select the view granularity (Function, Statement, Linked Object, or Loop), and then select the type of view. For the default views, select the "D" icon.

Statement in program that took the most time.

Statement Level Granularity Selected

The screenshot shows the OpenSpeedShop GUI with the following components:

- Process Control:** Run, Cont, Pause, Update, Terminate buttons.
- Status:** Process Loaded: Click on the "Run" button to begin the experiment.
- Stats Panel:** ManageProcessesPanel [1].
- View/Display Choice:** Functions, Statements (selected), Linked Objects, Loops.
- Executables:** smg2000 Hosts:(16) rzmerl1 ... Pids: 256 Rats: 256 Threads: 256
- Table:**

Exclusive CPU time in seconds.	Exclusive CPU time in	% of CPU Time	Statement Location (Line Number)
174.840000	-174.840000	28.436204	smg_residual.c(291)
57.530000	-57.530000	9.356754	cyclic_reduction.c(1130)
51.990000	-51.990000	8.455721	cyclic_reduction.c(910)
38.100000	-38.100000	6.196633	smg_residual.c(239)
35.290000	-35.290000	5.739611	smg_residual.c(147)
34.680000	-34.680000	5.640400	cyclic_reduction.c(998)
19.920000	-19.920000	3.239815	cyclic_reduction.c(753)
17.300000	-17.300000	2.813694	smg_residual.c(289)
other			
- Command Panel:** opens>>

You can manipulate the windows within the GUI and double click functions or statements to see the source code directly.

Double Click to open Source View and focus on source line (291)

Use window controls to split/arrange windows. Vertical split was used here.

The screenshot displays the OpenSpeedShop application interface. At the top, the title bar reads "Open|SpeedShop (on rzmerl156)". The interface is divided into several panels:

- Process Control:** Includes buttons for Run, Cont, Pause, Update, and Terminate. The status indicates "Process Loaded. Click on the 'Run' button to begin the experiment".
- Stats Panel [1]:** Shows "Showing Statements Report...". Below this is a table of executables and a detailed table of CPU time.
- Source Panel [1]:** Displays source code for "smg\_residual.c" with a vertical split. Line 291 is highlighted in red, and line 293 is highlighted in cyan. A double-click arrow points to line 291.
- Command Panel:** Contains a "ManageProcessesPanel [1]" with a table of process details.

Exclusive CPU time	% of CPU time	Statement Location (Line Num)
174.840000	28.436204	smg_residual.c(291)
-57.530000	9.356754	cyclic_reduction.c(1130)
-51.990000	8.455721	cyclic_reduction.c(910)
-38.100000	6.196633	smg_residual.c(239)
-35.290000	5.739611	smg_residual.c(147)
-34.680000	5.640400	cyclic_reduction.c(998)
-19.920000	3.239815	cyclic_reduction.c(753)

Processes:	Rank	Thread	Status	PID	Rank	Thread
-7537	224	46912523843552	Disconnected			
-7538	225	46912523843552	Disconnected			
-7539	226	46912523843552	Disconnected			

## 1.4 How to Gather and Understand Profiles

A profile is the aggregated measurements collected during the experiment. Profiles look at code sections over time. There are advantages to using profiles since they reduce the size of performance data and typically the data is collected with low overhead. So profiles can provide a good overview of the performance of an application.

The disadvantage of using a profile is that you are required to know beforehand how to aggregate the data collected. Also, since profiles provide more of an overview, they omit the performance details of individual events. There could also be an issue where selecting an inappropriate sampling frequency could skew the results of the profile.

Statistical Performance Analysis is a standard profiling technique; it involves interrupting the execution of the application in periodic intervals to record the location of the execution (Program Counter value). It can also be used to collect additional data like stack traces or hardware counters. Again the advantage of this method is its low overhead. It is good for getting an overview of the program and finding the hotspots (time intensive areas) within the program.

The sampling experiments available in O|SS include Program Counter Sampling, Call Path Profiling and Hardware Counter. The Program Counter Sampling experiment (osspsamp) provides approximate CPU time for each line and function in the program. The Call Path Profiling experiment (ossusertime) provides inclusive vs. exclusive CPU time (see section 4.2), and also includes call stacks. There is a number of Hardware Counter experiments (osshwc, osshwctime) that sample hardware counter overflows and osshwcsamp that can periodically sample up to six hardware counter events.

### 2.1 Program Counter Sampling (pcsamp) Experiment

A flat profile will answer the basic question: “Where does my code spend its time?” This will be displayed as a list of code elements with varying granularity, i.e. statements, functions and libraries (linked objects), with the time spent at each function. Flat profiling can be done through sampling, which allows us to avoid the overhead of direct measurements. We must ensure we request a sufficient number of samples (sampling rate) to get an accurate result.

O|SS can use this information to identify the critical regions. The profile shows computationally intensive code regions by displaying the time spent per function or per statement. While viewing this we must ask ourselves:

- “Are those the functions/statements that we expected to be taking the most time?”
- “Does this match the computational kernels?”
- “Are any runtime functions taking a lot of time?”

We want to identify any components that are bottlenecks. We can do this by viewing the profile aggregated by shared (linked) objects, making sure the correct or expected modules are present, then analyze the impact of those support and/or runtime libraries.

### 2.1.1 Program Counter Sampling (pcsamp) experiment performance data gathering

The program counter sampling experiment convenience script is “osspsamp”. Use this convenience script in this manner to gather address values for where OJSS periodically interrupted the application being run and took an address sample:

```
osspsamp “how you normally run your application” < sampling rate>
```

An example of flat profiling would be running the program counter sampling in OJSS. We will run the convenience script on our test program smg2000:

```
> osspsamp “mpirun -np 256 smg2000 -n 50 50 50”
```

It is recommended that you compile your code with the `-g` option in order to see the statements in the sampling. The pcsamp experiment also takes a sampling frequency as an optional parameter, the available parameters are high (200 samples per second), low (50 samples per second) and the default value is 100 samples per second. If we wanted to run the same experiment with the high sampling rate, we would simply issue the command:

```
> osspsamp “mpirun -np 256 smg2000 -n 50 50 50” high
```

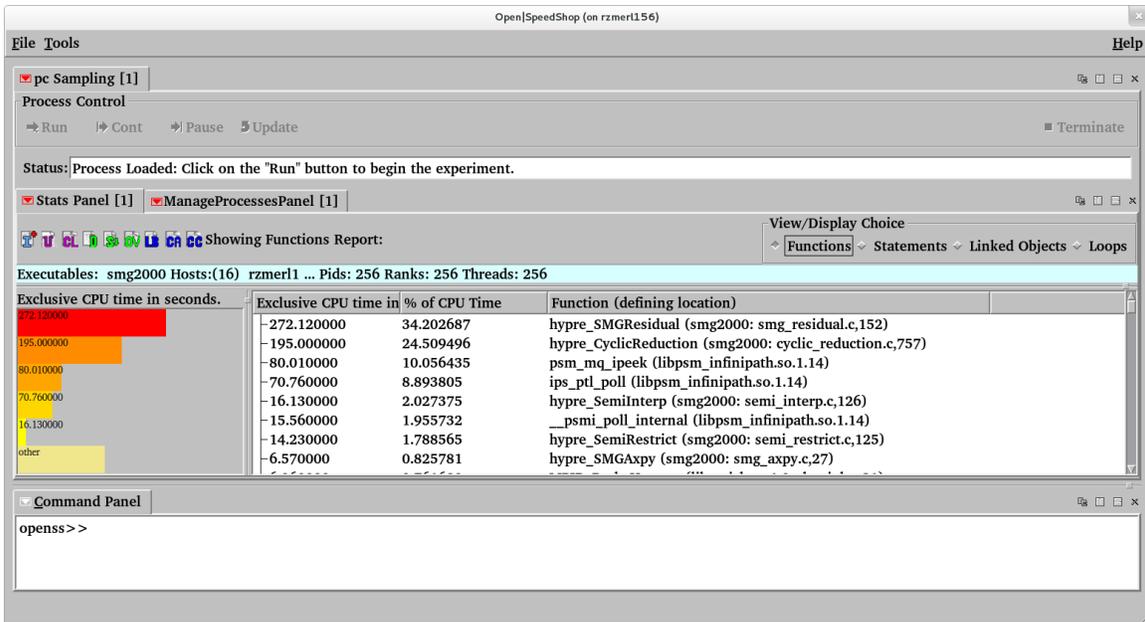
#### 2.1.1.1 Program Counter Sampling (pcsamp) experiment parameters

The pcsamp experiment is timer based. What that means is a timer is used to periodically interrupt the processor. For the pcsamp experiment, each time the timer interrupts the processor; the address in the program counter is read up and saved. This allows OJSS to map those address values back to the source when the pcsamp performance information is viewed via the command line interface (CLI) or the graphical user interface (GUI) tool.

In the next example the user is choosing to only sample 45 times a second instead of the default 100 times a second. Why would you want to do this? One reason would be to save database size; a lower sampling rate may still give an accurate portrayal of the application behavior.

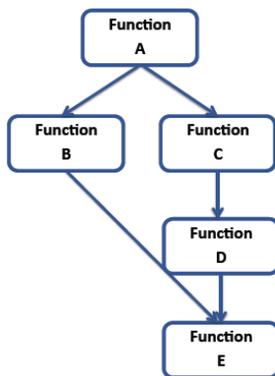
### 2.1.2 Program Counter Sampling (pcsamp) experiment performance data viewing with GUI

We can view the results of this flat profile in the O|SS GUI by using the “opens -f <database filename>” command.



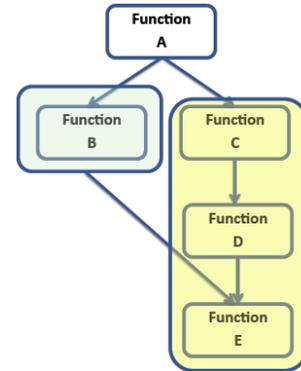
### 2.1.3 Program Counter Sampling (pcsamp) experiment performance data viewing with CLI

## 3.1 Call Path Profiling (usertime) Experiment



The call path profiling (usertime) experiment can add some information that is missing from the flat profiles. It is able to distinguish routines called from multiple callers, and understand the call invocation history. This provides context for the performance data. It also gathers stack traces for each performance sample and only aggregates samples with equal stack traces. For the user, this simplifies the view by showing the caller/callee relationship. It can also highlight the hot call paths, the paths through the application that take the most time.

The call path profiling experiment also provides inclusive and exclusive time. Exclusive time is the time spent inside a function only, for example function B. Whereas inclusive time is the time spent inside a function and its children, for example the full chain of function C, D and E.



The call path profiling experiment is similar to the program counter sampling experiment since it collects program counter information, except that it collects call stack information at every sample. There are, of course, tradeoffs with that, you obtain additional context information from the call stacks but there is now a higher overhead and necessarily lower sampling rate.

### 3.1.1 Call Path Profiling (usertime) experiment performance data gathering

We can run the call path profiling experiment using the O|SS convenience script on our test program smg2000:

```
> ossusertime "mpirun -np 256 smg2000 -n 50 50 50"
```

Again it is recommended that you compile your code with the `-g` option in order to see the statements in the sampling. The `usertime` experiment also takes a sampling frequency as an optional parameter, the available parameters are high (70 samples per second), low (18 samples per second) and the default value is 35 samples per second. Note that these sample rates are lower than the `pcsamp` experiment because of the increased amount of data being collected. If we wanted to run the same experiment with the low sampling rate, we would simply issue the command:

```
> ossusertime "mpirun -np 256 smg2000 -n 50 50 50" low
```

Here is an example run of an `usertime` experiment with full output:

```
ossusertime "mpirun -np 4 ./smg2000 -n 65 65 65"
[openss]: usertime experiment using the default sampling rate: "35".
Creating topology file for frontend host localhost
Generated topology file: ./cbtfAutoTopology
Running usertime collector.
Program: mpirun -np 4 ./smg2000 -n 65 65 65
Number of mrnet backends: 4
Topology file used: ./cbtfAutoTopology
executing mpi program: mpirun -np 4 cbtfrun --mpi --mrnet -c usertime ./smg2000 -n 65 65 65
Running with these driver parameters:
  (nx, ny, nz) = (65, 65, 65)
  (Px, Py, Pz) = (4, 1, 1)
  (bx, by, bz) = (1, 1, 1)
  (cx, cy, cz) = (1.000000, 1.000000, 1.000000)
  (n_pre, n_post) = (1, 1)
  dim = 3
  solver ID = 0
=====
Struct Interface:
=====
```

```

Struct Interface:
  wall clock time = 0.023957 seconds
  cpu clock time = 0.030000 seconds
=====
Setup phase times:
=====
SMG Setup:
  wall clock time = 0.594738 seconds
  cpu clock time = 0.590000 seconds
=====
Solve phase times:
=====
SMG Solve:
  wall clock time = 4.306247 seconds
  cpu clock time = 4.280000 seconds

Iterations = 7
Final Relative Residual Norm = 1.760588e-07

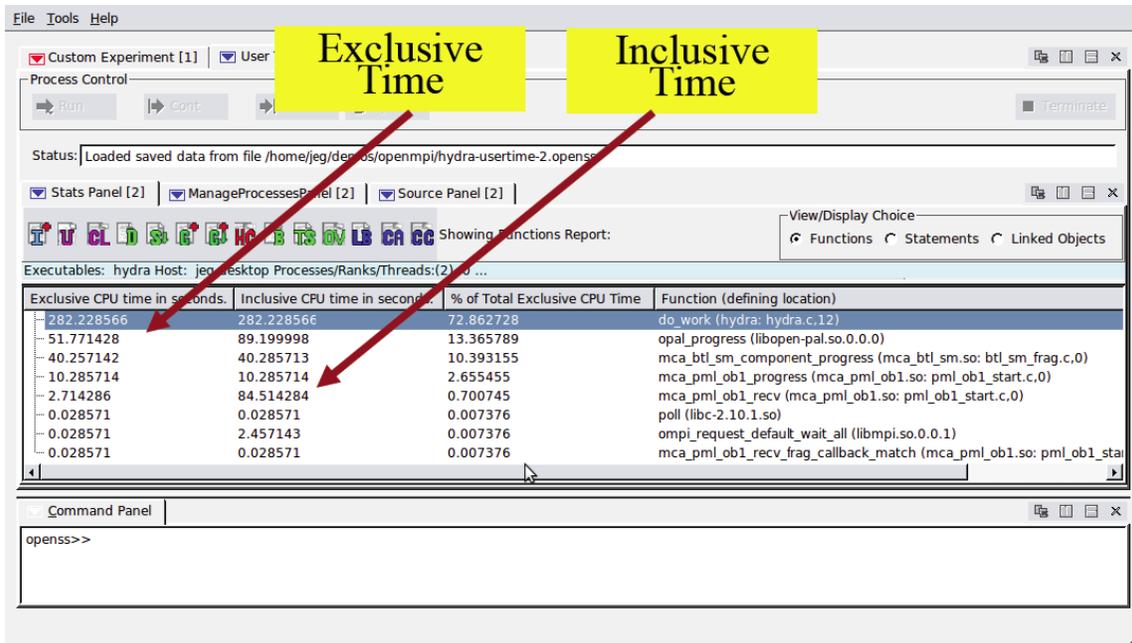
All Threads are finished.
default view for ./smg2000-usertime-14.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 5.271756 seconds from 2016/11/11 08:06:12 to 2016/11/11 08:06:17

Exclusive Inclusive  % of Function (defining location)
CPU time  CPU time  Total
  in      in Exclusive
seconds. seconds. CPU Time
9.000000  9.800000  45.718433  hypre_SMGResidual (smg2000: smg_residual.c,152)
4.171428  7.057143  21.190131  hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
0.542857  0.571429  2.757620  hypre_SemiInterp (smg2000: semi_interp.c,126)
0.514286  1.542857  2.612482  mca_btl_vader_check_fboxes (libmpi.so.12.0.2: btl_vader_fbox.h,184)
0.514286  0.942857  2.612482  pack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_pack.h,35)
0.485714  0.485714  2.467344  __memcpy_ssse3_back (libc-2.17.so)
0.457143  0.514286  2.322206  unpack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_unpack.h,34)
0.314286  2.314286  1.596517  opal_progress (libopen-pal.so.13.0.2: opal_progress.c,151)
0.285714  0.314286  1.451379  hypre_SemiRestrict (smg2000: semi_restrict.c,125)
0.257143  0.257143  1.306241  hypre_StructAxy (smg2000: struct_axpy.c,25)
0.228571  0.228571  1.161103  hypre_SMGAxy (smg2000: smg_axpy.c,27)
0.171429  0.171429  0.870827  hypre_SMGSetStructVectorConstantValues (smg2000: smg.c,379)

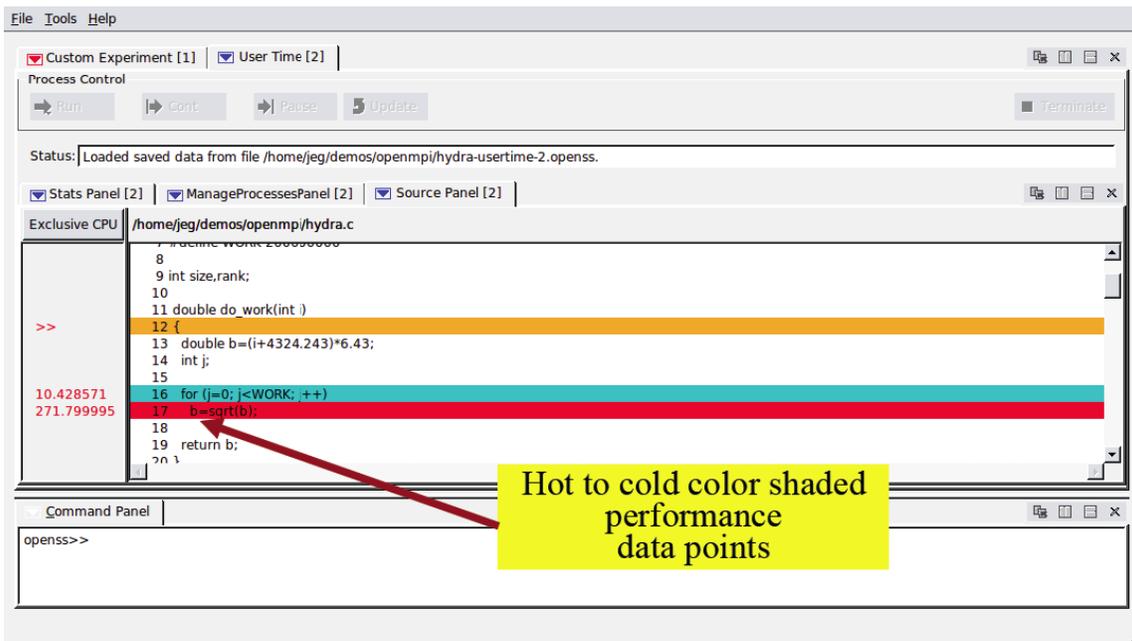
```

### 3.1.2 Call Path Profiling (usertime) experiment performance data viewing with GUI

We can view the results of this experiment in the O|SS GUI. The view is similar to the pcsamp view but this time the inclusive CPU time is also shown.

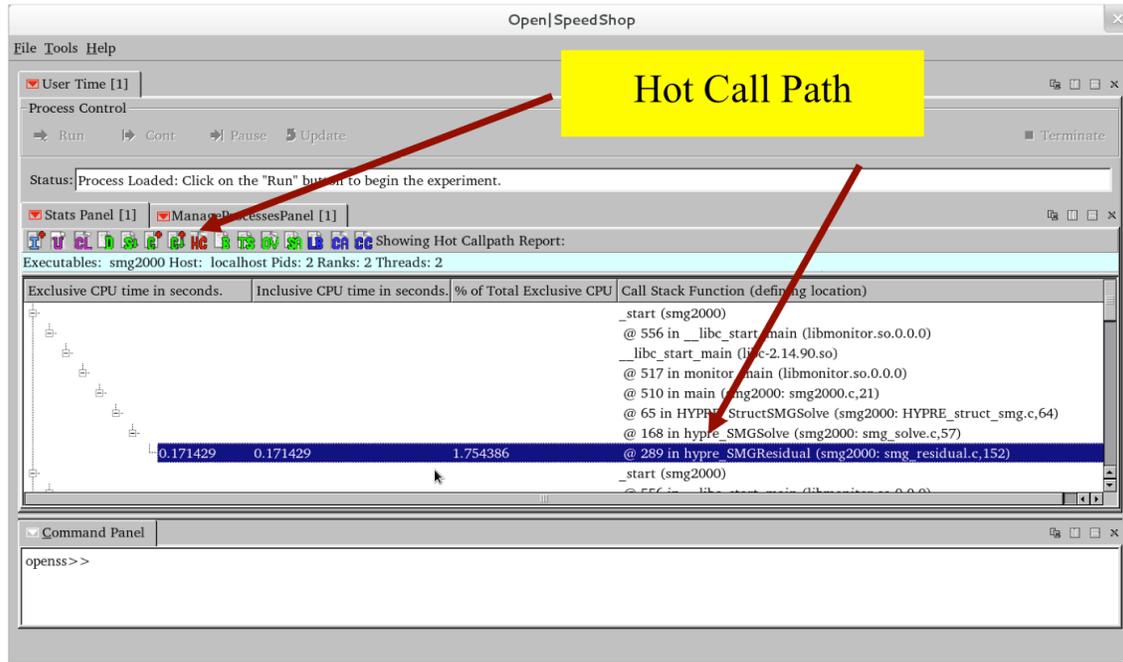


Below we see the Exclusive CPU time on highlighted lines that indicate relatively high CPU times.

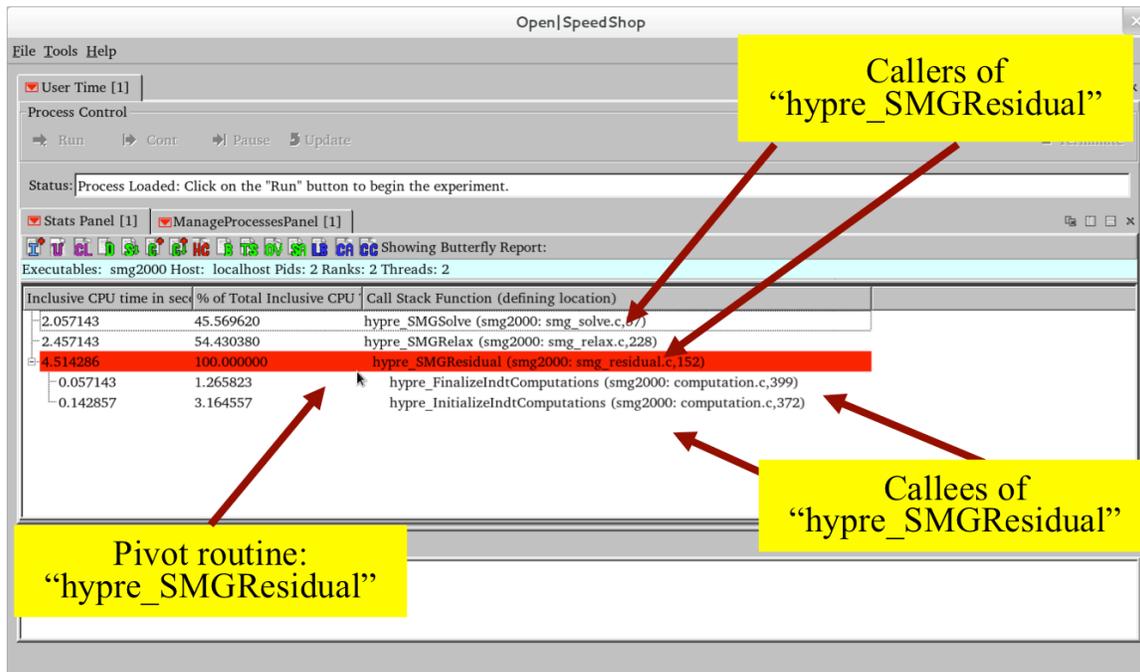


While performance tools will point out potential bottlenecks and hot areas it is still up to the user to interpret most data in the correct context as well as note areas of the code you may want to probe further. If the inclusive and exclusive times are similar this means the child executions are insignificant (with respect to CPU time) and it may not be useful to profile below this layer. If the inclusive time is

significantly greater than the exclusive time, then you should focus your attention to the execution times of the children.



The stack trace views in O|SS are similar to the well-known Unix profiling tool gprof.



### 3.1.3 Call Path Profiling (usertime) experiment performance data viewing with CLI

The table below describes information that is included in the usertime experiment default view.

Column Name	Column Definition
Exclusive CPU Time	Aggregated total exclusive time spent in the application function corresponding to this row of data.
% of CPU Time	Percentage of exclusive time spent in the function corresponding to this row of data relative to the total application exclusive time for all the application functions.
Inclusive CPU Time	Aggregated total inclusive time spent in the application function corresponding to this row of data.

To load a database file into the CLI, use this form of the openss client:

```
$ openss -cli -f ./smg2000-usertime-14.openss
```

This is a default CLI view for the usertime experiment, restricted to the top 10 time taking functions. Using the experiment name and the number of items that are desired to be displayed limits the output to those items.

```
openss>>expview usertime10
```

```
Exclusive Inclusive   % of Function (defining location)
CPU time  CPU time   Total
  in      in Exclusive
seconds. seconds. CPU Time
9.000000  9.800000  45.718433  hypre_SMGResidual (smg2000: smg_residual.c,152)
4.171428  7.057143  21.190131  hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
0.542857  0.571429  2.757620  hypre_SemiInterp (smg2000: semi_interp.c,126)
0.514286  1.542857  2.612482  mca_btl_vader_check_fboxes (libmpi.so.12.0.2:
btl_vader_fbox.h,184)
0.514286  0.942857  2.612482  pack_predefined_data (libopen-pal.so.13.0.2:
opal_datatype_pack.h,35)
0.485714  0.485714  2.467344  __memcpy_ssse3_back (libc-2.17.so)
0.457143  0.514286  2.322206  unpack_predefined_data (libopen-pal.so.13.0.2:
opal_datatype_unpack.h,34)
0.314286  2.314286  1.596517  opal_progress (libopen-pal.so.13.0.2: opal_progress.c,151)
0.285714  0.314286  1.451379  hypre_SemiRestrict (smg2000: semi_restrict.c,125)
0.257143  0.257143  1.306241  hypre_StructAxy (smg2000: struct_axpy.c,25)
```

The display below shows the top 10 time taking statements in the program:

```
openss>>expview -v statements usertime10
```

```
Exclusive Inclusive   % of Statement Location (Line Number)
CPU time  CPU time   Total
```

in	in Exclusive	CPU Time	
seconds.	seconds.		
7.085714	7.085714	41.471572	smg_residual.c(289)
1.371429	1.371429	8.026756	cyclic_reduction.c(1130)
1.314286	1.314286	7.692308	smg_residual.c(238)
0.828571	0.828571	4.849498	cyclic_reduction.c(910)
0.485714	0.485714	2.842809	cyclic_reduction.c(999)
0.285714	0.285714	1.672241	smg_residual.c(287)
0.285714	0.285714	1.672241	cyclic_reduction.c(853)
0.285714	0.285714	1.672241	cyclic_reduction.c(1000)
0.257143	0.257143	1.505017	semi_interp.c(294)
0.228571	0.228571	1.337793	opal_datatype_unpack.h(59)

The top 10 time consuming loops in the application are shown below. The line numbers (for example: 204 in first entry) is the line that the loop begins. Our static analysis does not provide us with the ending line number for the loop, but it is the line that corresponds to the end of the logical loop construct.

```

openss>>expview -v loops usertime10

Exclusive Inclusive   % of Loop Start Location (Line Number)
CPU time  CPU time   Total
  in      in Exclusive
seconds. seconds. CPU Time
8.971428  9.771428  37.649880 smg_residual.c(204)
1.914286  3.257143  8.033573  cyclic_reduction.c(1022)
1.885714  3.400000  7.913669  cyclic_reduction.c(882)
0.857143  0.885714  3.597122  cyclic_reduction.c(889)
0.485714  1.514286  2.038369  btl_vader_fbox.h(188)
0.400000  0.828571  1.678657  opal_datatype_pack.h(59)
0.371429  0.428571  1.558753  opal_datatype_unpack.h(58)
0.371429  0.400000  1.558753  semi_interp.c(238)
0.371429  0.400000  1.558753  cyclic_reduction.c(835)
0.371429  0.400000  1.558753  semi_interp.c(258)
openss>>

```

The time spent in the libraries of the application are shown here:

```

openss>>expview -v linkedobjects

Exclusive Inclusive   % of LinkedObject
CPU time  CPU time   Total
  in      in Exclusive
seconds. seconds. CPU Time
15.514285 19.742857 78.581766 smg2000
1.800000  3.400000  9.117221  libopen-pal.so.13.0.2
1.400000  3.885714  7.091172  libmpi.so.12.0.2
1.028571 19.742857  5.209841  libc-2.17.so

```

The top three time consuming call paths in the application are shown here:

```
openss>>expview -v fullstack usertime3
```

```
Exclusive Inclusive   % of Call Stack Function (defining location)
CPU time  CPU time   Total
in      in Exclusive
seconds. seconds. CPU Time
    _start (smg2000)
    > @ 556 in _libc_start_main (libmonitor.so.0.0.0: main.c,541)
    >> _libc_start_main (libc-2.17.so)
    >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
    >>>> @ 510 in main (smg2000: smg2000.c,21)
    >>>>> @ 65 in HYPRE_StructSMGSolve (smg2000: HYPRE_struct_smg.c,64)
    >>>>>> @ 224 in hypre_SMGSolve (smg2000: smg_solve.c,57)
0.800000 0.800000 4.052098 >>>>>> @ 289 in hypre_SMGResidual (smg2000:
smg_residual.c,152)
    _start (smg2000)
    > @ 556 in _libc_start_main (libmonitor.so.0.0.0: main.c,541)
    >> _libc_start_main (libc-2.17.so)
    >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
    >>>> @ 510 in main (smg2000: smg2000.c,21)
    >>>>> @ 65 in HYPRE_StructSMGSolve (smg2000: HYPRE_struct_smg.c,64)
    >>>>>> @ 168 in hypre_SMGSolve (smg2000: smg_solve.c,57)
0.400000 0.400000 2.026049 >>>>>> @ 289 in hypre_SMGResidual (smg2000:
smg_residual.c,152)
    _start (smg2000)
    > @ 556 in _libc_start_main (libmonitor.so.0.0.0: main.c,541)
    >> _libc_start_main (libc-2.17.so)
    >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
    >>>> @ 510 in main (smg2000: smg2000.c,21)
    >>>>> @ 65 in HYPRE_StructSMGSolve (smg2000: HYPRE_struct_smg.c,64)
    >>>>>> @ 164 in hypre_SMGSolve (smg2000: smg_solve.c,57)
    >>>>>>> @ 325 in hypre_SMGRelax (smg2000: smg_relax.c,228)
    >>>>>>>> @ 224 in hypre_SMGSolve (smg2000: smg_solve.c,57)
0.400000 0.400000 2.026049 >>>>>>>> @ 289 in hypre_SMGResidual (smg2000:
smg_residual.c,152)
```

Here the Butterfly view shows the functions calling hypre\_SMGSolve and also the functions hypre\_SMGSolve calls. Hypre\_SMGSolve is the pivot point in this view.

```
openss>>expview -vbutterfly -f hypre_SMGSolve
```

```
Inclusive % of Total Call Stack Function (defining location)
CPU time Inclusive
in CPU Time
seconds.
17.200000 94.654088 <HYPRE_StructSMGSolve (smg2000: HYPRE_struct_smg.c,64)
0.971429 5.345912 <hypre_SMGRelax (smg2000: smg_relax.c,228)
18.171428 100.000000 hypre_SMGSolve (smg2000: smg_solve.c,57)
16.285714 89.622642 >hypre_SMGRelax (smg2000: smg_relax.c,228)
1.428571 7.861635 >hypre_SMGResidual (smg2000: smg_residual.c,152)
0.171429 0.943396 >hypre_SemiInterp (smg2000: semi_interp.c,126)
0.114286 0.628931 >hypre_SemiRestrict (smg2000: semi_restrict.c,125)
0.114286 0.628931 >hypre_StructApxy (smg2000: struct_axpy.c,25)
```

```
0.057143 0.314465 >hypre_StructInnerProd (smg2000: struct_innerprod.c,32)
```

Here the performance information for only percent is displayed because the metric percent was given to the expview command.

```
openss>>expview -m percent usertime9
```

```
% of Function (defining location)
Total
Exclusive
Time
45.718433 hypre_SMGResidual (smg2000: smg_residual.c,152)
21.190131 hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
2.757620 hypre_SemiInterp (smg2000: semi_interp.c,126)
2.612482 mca_btl_vader_check_fboxes (libmpi.so.12.0.2: btl_vader_fbox.h,184)
2.612482 pack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_pack.h,35)
2.467344 __memcpy_ssse3_back (libc-2.17.so)
2.322206 unpack_predefined_data (libopen-pal.so.13.0.2: opal_datatype_unpack.h,34)
1.596517 opal_progress (libopen-pal.so.13.0.2: opal_progress.c,151)
1.451379 hypre_SemiRestrict (smg2000: semi_restrict.c,125)
```

## 4 How to Relate Data to Architectural Properties

Performance Application Programming Interface (PAPI) allows access to hardware counters through APIs and simple runtime tools. You can find more information on PAPI at <http://icl.cs.utk.edu/papi>.

OJSS provides three hardware counter experiments that are implemented on top of PAPI. It provides access to PAPI and native counters like data cache misses, TLB misses and bus accesses.

There are a few basic models to follow in hardware counter experiments. The first is thresholding, where the user selects a counter and the application runs until a fixed number of events have been reached on that counter. Then a PC sample is taken at that location every time the counter increases by the preset fixed number. The ideal threshold, the fixed number at which to monitor, is dependent on the application. Another model is a timer based sampling where the counters are checked at given time intervals.

OJSS provides three hardware counter experiments, hwc for flat hardware counter profiles using a single hardware counter, hwctime for profiles with stack traces using a single hardware counter and hwcsamp for PC sampling with multiple hardware counters. Both osshwc and osshwctime support non-derived PAPI presets, all non-derived events are reported by “papi\_avail -a”. You can also see the available events by running the experiments (osshwc or osshwctime) with no arguments. The experiments include all native events for that specific architecture. Some PAPI event names are listed in sections below, but please see the PAPI documentation for the full list.

The threshold you choose depends on the application, you want to balance overhead with accuracy. Remember a higher threshold will record less samples. Rare events need a smaller threshold or that information may be lost (never triggered and recorded). Frequent events should use a larger threshold, to reduce the overhead of collecting the information. Selecting the right threshold can take experience or some trial and error.

**HINT:** Running the sampling based hardware counter experiment, osshwcsamp, can help you get an idea for a threshold value to try when running the osshwc and osshwctime experiments which are threshold based. Since the ideal number of events (threshold) depends on the application and the selected counter, for events other than the default, the hwcsamp experiment can be used to get an overview of counter activity.

The default threshold is set to a very large value to match the default event (PAPI\_TOT\_CYC). For all other events, it is recommended that the user run hwcsamp first to get an idea of how many times a particular event occurs (the count

of the event) during the life of the program. A reasonable threshold can be determined from the hwcsamp data by determining the average counts per thread of execution and then setting the hwc/hwctime threshold to some small fraction of that. For example, if you see 1333333333 PAPI\_L1\_DCM's over the life of the program when running the hwcsamp experiment and there were 524 processes used during the application run, then this is the formula you could use to find a reasonable threshold for the hwc and hwctime experiments when using the PAPI\_L1\_DCM event for the same application. So the formula that could be used is as follows:

```
(Average counts per thread) / 1000 == Threshold for hwc/hwctime
In this case:
(1333333333/524)/1000 == 2544529/1000 == 2545
```

Using this formula one could use 2545 as the threshold value in hwc and hwctime for PAPI\_L1\_DCM and expect to get a reasonable data sample of that event.

**NOTE:** The number of PAPI counters and their use can be overwhelming. Ratios derived from a combination of hardware events can sometimes provide more useful information than raw metrics. Develop the ability to interpret metric ratios with a focus on understanding:

- Instructions per cycle or cycles per instruction
- Floating point / Vectorization efficiency
- Cache behaviors; Long latency instruction impact
- Branch mispredictions
- Memory and resource access patterns
- Pipeline stalls

#### 4.1 Hardware Counter Experiment (hwc)

As an example we will run the osshwc experiment on our test program smg2000. The convenience script for this is experiment is:

```
> osshwc "mpirun -np 256 smg2000 -n 50 50 50" <counter> <threshold>
```

This is the same syntax as the osshwctime experiment. Note that if your output is empty, try lowering the <threshold> value, it is calculated by O|SS by default. You can try lowering the threshold value if there have not been enough PAPI event occurrences to record. Also see the HINT in the osshwcsamp section above. You can run osshwcsamp and use a formula to create a reasonable threshold. Any counter reported by "papi\_avail -a" that is not derived is available for use. You can also see the available counters by using the osshwc or osshwctime commands with no arguments. Native counters are listed in the PAPI documentation.

PAPI Name	Description	Threshold
PAPI_L1_DCM	L1 data cache misses	high
PAPI_L2_DCM	L2 data cache misses	high/medium
PAPI_L1_DCA	L1 data cache accesses	high
PAPI_FPU_IDL	Cycles in which FPUs are idle	high/medium
PAPI_STL_ICY	Cycles with no instruction issue	high/medium
PAPI_BR_MSP	Miss-predicted branches	medium/low
PAPI_FP_INS	Number of floating point instructions	high
PAPI_LD_INS	Number of load instructions	high
PAPI_VEC_INS	Number of vector/SIMD instructions	high/medium
PAPI_HW_INT	Number of hardware interrupts	low
PAPI_TLB_TL	Number of TLB misses	low

Note the Threshold indications are just for rough guidance and are dependent on the application. Also remember that not all counters will exist on all platforms, run `osshwc` with no arguments to see the available hardware counters available.

In the sections below, we show the outputs from the `osshwc` experiment, note that the default counter is the total cycles.

#### 4.1.1 Hardware Counter Threshold (hwc) experiment performance data gathering

The hardware counter threshold experiment convenience script is “`osshwc`”. Use this convenience script in this manner to gather counter values for one unique hardware counter:

```
osshwc "how you normally run your application" <papi event > < threshold value>
```

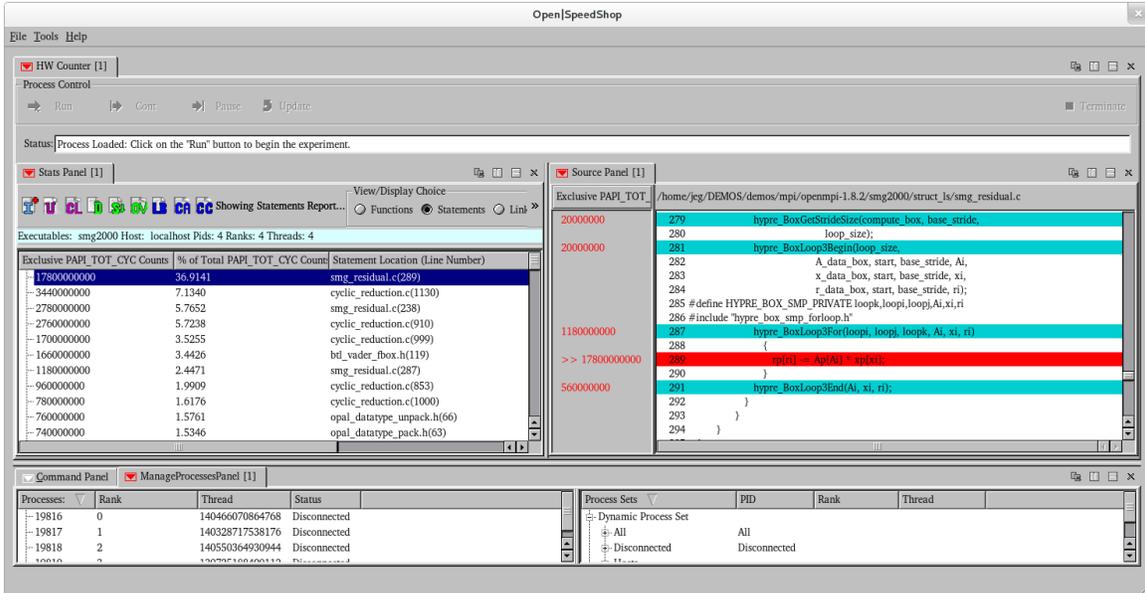
The following is an example of how to gather data for the `smg2000` application on a Linux cluster platform using the `osshwc` convenience script. It gathers performance data for the default counter, `PAPI_TOT_CYC` because there is no hardware counter value specified after the quoted application run command.

```
osshwc "mpirun -np 4 ./smg2000 -n 60 60 60"
```

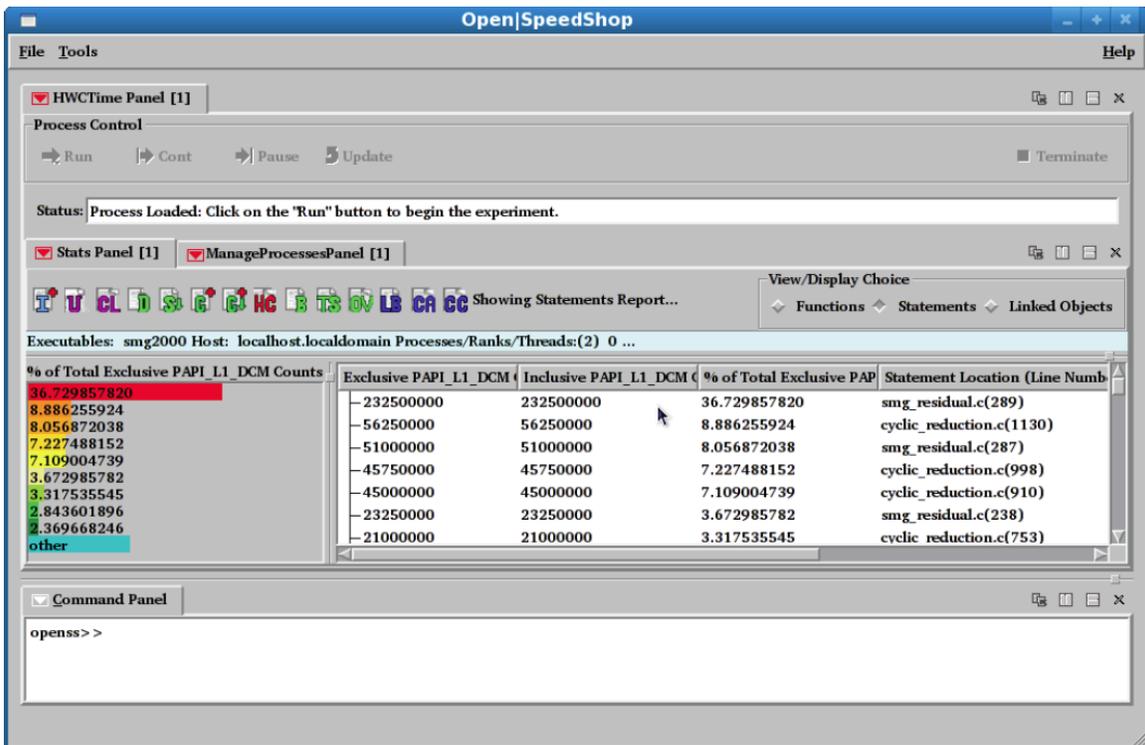
#### 4.1.2 Hardware Counter Threshold (hwc) experiment performance data viewing with GUI

To launch the GUI on any experiment, use “`openss -f <database name>`”.

This image shows the default view for the hwc experiment run with the smg2000 MPI application using PAPI\_TOT\_CYC as the hardware counter event. Double clicking on a line in the Stats Panel or on the bar chart will take the user to the source file and line represented by that line of performance information.



The next image displays the output from the osshwctime experiment where the counter is the L1 cache misses.



### 4.1.3 Hardware Counter Threshold (hwc) experiment performance data viewing with CLI

To launch the CLI on any experiment, use “`openss -cli -f <database name>`”. In this example we show three view default CLI views with different granularities: function, statement, and library level.

```
openss -f smg2000-hwc-3.openss
[openss]: The restored experiment identifier is: -x 1

[jeg@localhost test]$ openss -cli -f smg2000-hwc-3.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview
Exclusive      % of Total      Function (defining location)
PAPI_TOT_CYC  PAPI_TOT_CYC
Counts        Counts
23080000000   43.8283   hypre_SMGResidual (smg2000: smg_residual.c,152)
12880000000   24.4588   hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
35400000000   6.7224    mca_btl_vader_check_fboxes (libmpi.so.1.5.2: btl_vader_fbox.h,106)
14200000000   2.6965    unpack_predefined_data (libopen-pal.so.6.2.0: opal_datatype_unpack.h,41)
12200000000   2.3167    hypre_SemiInterp (smg2000: semi_interp.c,126)
11400000000   2.1648    pack_predefined_data (libopen-pal.so.6.2.0: opal_datatype_pack.h,38)
10200000000   1.9370    __memcpy_ssse3_back (libc-2.17.so)
7400000000    1.4052    hypre_SemiRestrict (smg2000: semi_restrict.c,125)
...
...

openss>>expview -v statements

Exclusive      % of Total      Statement Location (Line Number)
PAPI_TOT_CYC  PAPI_TOT_CYC
Counts        Counts
17800000000   36.9141   smg_residual.c(289)
34400000000   7.1340    cyclic_reduction.c(1130)
27800000000   5.7652    smg_residual.c(238)
27600000000   5.7238    cyclic_reduction.c(910)
17000000000   3.5255    cyclic_reduction.c(999)
16600000000   3.4426    btl_vader_fbox.h(119)
11800000000   2.4471    smg_residual.c(287)
9600000000    1.9909    cyclic_reduction.c(853)
...
...

openss>>expview -v linkedobjects

Exclusive      % of Total      LinkedObject
PAPI_TOT_CYC  PAPI_TOT_CYC
Counts        Counts
40800000000   77.3606    smg2000
60600000000   11.4903    libmpi.so.1.5.2
41600000000   7.8878     libopen-pal.so.6.2.0
17200000000   3.2613     libc-2.17.so
```

## 4.2 Hardware Counter Time Experiment (hwctime)

As an example we will run the `osshwc` experiment on our test program `smg2000`. The convenience script for this is `experiment` is:

```
> osshwctime "mpirun -np 256 smg2000 -n 50 50 50" <counter> <threshold>
```

This is the same syntax as the `osshwc` experiment. Note that if your output is empty, try lowering the `<threshold>` value, it is calculated by `OJSS` by default. You can try lowering the threshold value if there have not been enough PAPI event occurrences to record. Also see the HINT in the `osshwcsamp` section below. You can run `osshwcsamp` and use a formula to create a reasonable threshold. Any counter reported by “`papi_avail -a`” that is not derived is available for use. You can also see the available counters by using the `osshwc` or `osshwctime` commands with no arguments. Native counters are listed in the PAPI documentation.

PAPI Name	Description	Threshold
PAPI_L1_DCM	L1 data cache misses	high
PAPI_L2_DCM	L2 data cache misses	high/medium
PAPI_L1_DCA	L1 data cache accesses	high
PAPI_FPU_IDL	Cycles in which FPUs are idle	high/medium
PAPI_STL_ICY	Cycles with no instruction issue	high/medium
PAPI_BR_MSP	Miss-predicted branches	medium/low
PAPI_FP_INS	Number of floating point instructions	high
PAPI_LD_INS	Number of load instructions	high
PAPI_VEC_INS	Number of vector/SIMD instructions	high/medium
PAPI_HW_INT	Number of hardware interrupts	low
PAPI_TLB_TL	Number of TLB misses	low

Note the Threshold indications are just for rough guidance and are dependent on the application. Also remember that not all counters will exist on all platforms, run `osshwc` with no arguments to see the available hardware counters available.

In the sections below, we show the outputs from the `osshwctime` experiment, note that the default counter is the total cycles.

#### 4.2.1 Hardware Counter Time Threshold (hwctime) experiment performance data gathering

The hardware counter threshold experiment convenience script is “osshwc”. Use this convenience script in this manner to gather counter values for one unique hardware counter:

```
osshwctime “how you normally run your application” <papi event > < threshold value>
```

The following is an example of how to gather data for the smg2000 application on a Linux cluster platform using the osshwc convenience script. The osshwctime convenience script will gather performance data for the default counter, PAPI\_TOT\_CYC if there is no hardware counter value specified after the quoted application run command. In our example, we specify an alternative counter, PAPI\_L1\_DCM and a specific threshold value, 750000. Each time the threshold value is reached, a sample will be taken and recorded. At program completion, an O|SS database file is created and the performance data can be viewed. A default report is shown as part of the O|SS convenience script (see below).

```
$ osshwctime "mpirun -np 4 ./smg2000 -n 65 65 65" PAPI_L1_DCM 750000
[openss]: hwctime using default threshold: 750000.
[openss]: hwctime using user specified papi event: "PAPI_L1_DCM"
Creating topology file for frontend host localhost
Generated topology file: ./cbtfAutoTopology
Running hwctime collector.
Program: mpirun -np 4 ./smg2000 -n 65 65 65
Number of mrnet backends: 4
Topology file used: ./cbtfAutoTopology
executing mpi program: mpirun -np 4 cbtfrun --mpi --mrnet -c hwctime ./smg2000 -n 65
65 65
Running with these driver parameters:
(nx, ny, nz) = (65, 65, 65)
(Px, Py, Pz) = (4, 1, 1)
(bx, by, bz) = (1, 1, 1)
(cx, cy, cz) = (1.000000, 1.000000, 1.000000)
(n_pre, n_post) = (1, 1)
dim = 3
solver ID = 0
=====
Struct Interface:
=====
Struct Interface:
wall clock time = 0.024858 seconds
cpu clock time = 0.030000 seconds
=====
Setup phase times:
=====
SMG Setup:
wall clock time = 0.624002 seconds
cpu clock time = 0.620000 seconds
```

```

=====
Solve phase times:
=====
SMG Solve:
  wall clock time = 3.907005 seconds
  cpu clock time  = 3.870000 seconds

Iterations = 7
Final Relative Residual Norm = 1.760588e-07

All Threads are finished.
default view for ./smg2000-hwctime-4.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 4.818158 seconds from 2016/11/11 11:12:31 to 2016/11/11
11:12:36

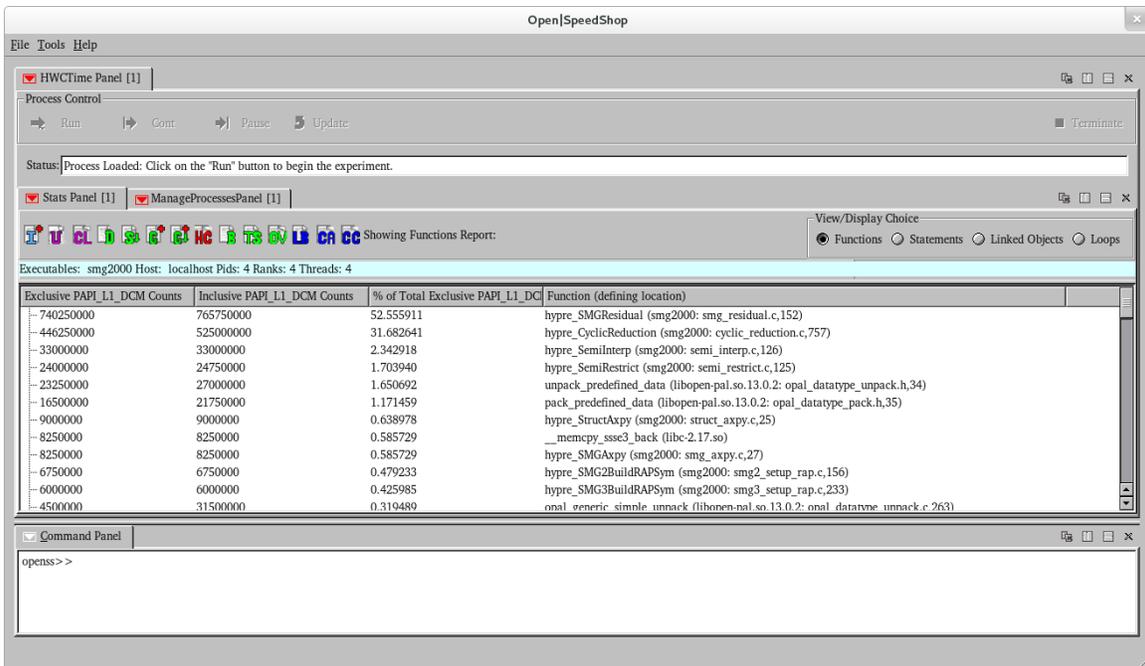
Exclusive Inclusive % of Total Function (defining location)
PAPI_L1_DCM PAPI_L1_DCM Exclusive
Counts Counts PAPI_L1_DCM
Counts
740250000 765750000 52.555911 hypre_SMGResidual (smg2000: smg_residual.c,152)
446250000 525000000 31.682641 hypre_CyclicReduction (smg2000:
cyclic_reduction.c,757)
330000000 330000000 2.342918 hypre_SemiInterp (smg2000: semi_interp.c,126)
240000000 247500000 1.703940 hypre_SemiRestrict (smg2000: semi_restrict.c,125)
232500000 270000000 1.650692 unpack_predefined_data (libopen-pal.so.13.0.2:
opal_datatype_unpack.h,34)
165000000 217500000 1.171459 pack_predefined_data (libopen-pal.so.13.0.2:
opal_datatype_pack.h,35)
90000000 90000000 0.638978 hypre_StructAxy (smg2000: struct_axpy.c,25)
82500000 82500000 0.585729 hypre_SMGAxy (smg2000: smg_axpy.c,27)
82500000 82500000 0.585729 __memcpy_sse3_back (libc-2.17.so)
67500000 67500000 0.479233 hypre_SMG2BuildRAPSym (smg2000:
smg2_setup_rap.c,156)
60000000 60000000 0.425985 hypre_SMG3BuildRAPSym (smg2000:
smg3_setup_rap.c,233)

```

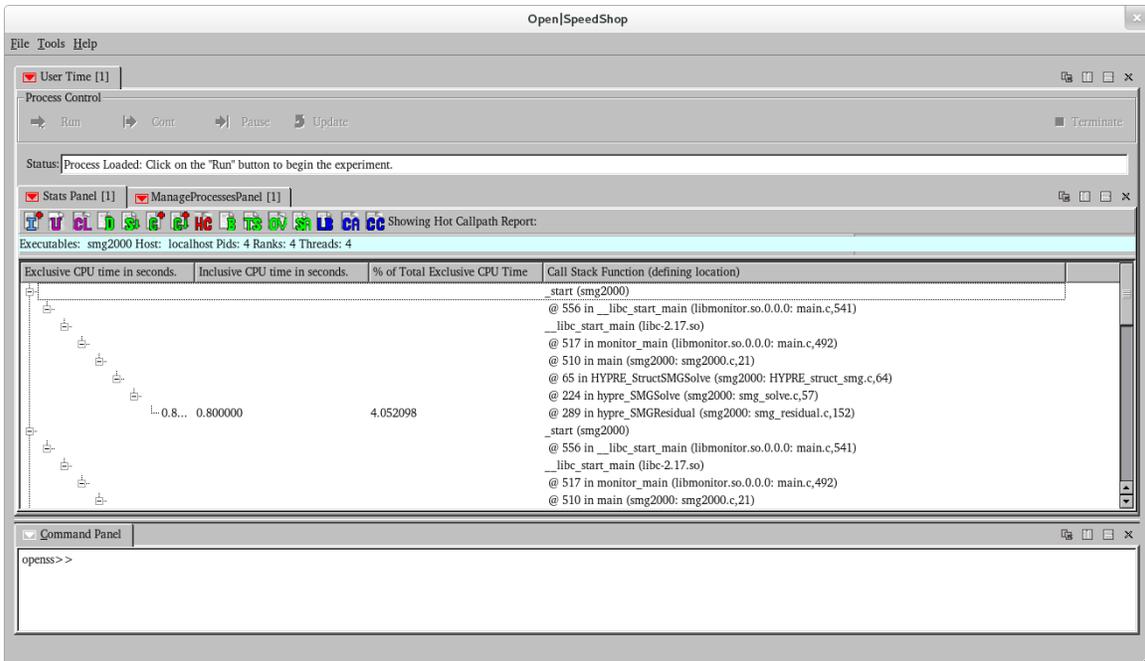
#### 4.2.2 Hardware Counter Threshold (hwctime) experiment performance data viewing with GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

This image shows the default view for the hwc experiment run with the smg2000 MPI application using PAPI\_L1\_DCM as the hardware counter event. Double clicking on a line in the Stats Panel or on the bar chart will take the user to the source file and line represented by that line of performance information.

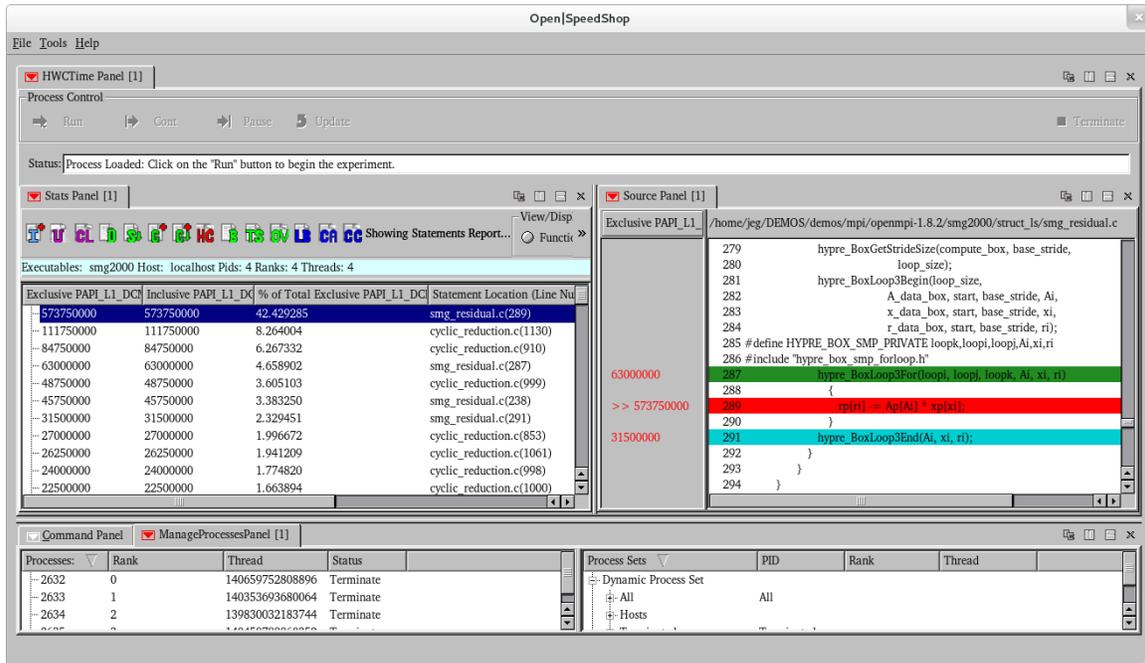


The next image displays the output from the `osshwctime` experiment where the Hot Call Path icon (HC in red) was chosen. This displays the top 5 time taking call paths in the `smg2000` application.



The view below shows the top time taking statements and the source panel focused on the statement in `smg2000` that took the most time. Double clicking on the

statistics lines in the StatsPanel will focus the source panel on the source line corresponding to the statistics line.



#### 4.2.3 Hardware Counter Time Threshold (hwctime) experiment performance data viewing with CLI

To launch the CLI on any experiment, use “opens -cli -f <database name>”. In this example we show three view default CLI views with different granularities: function, statement, and library level.

This is the CLI default view for the hwctime experiment.

```
[jeg@localhost test]$ opens -cli -f smg2000-hwctime-4.opens
opens>>[opens]: The restored experiment identifier is: -x 1
opens>>expview hwctime10
```

```
Exclusive Inclusive % of Total Function (defining location)
PAPI_L1_DCM PAPI_L1_DCM Exclusive
Counts Counts PAPI_L1_DCM
Counts
740250000 765750000 52.555911 hypre_SMGResidual (smg2000: smg_residual.c,152)
446250000 525000000 31.682641 hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
33000000 33000000 2.342918 hypre_SemiInterp (smg2000: semi_interp.c,126)
24000000 24750000 1.703940 hypre_SemiRestrict (smg2000: semi_restrict.c,125)
23250000 27000000 1.650692 unpack_predefined_data (libopen-pal.so.13.0.2:
opal_datatype_unpack.h,34)
16500000 21750000 1.171459 pack_predefined_data (libopen-pal.so.13.0.2:
opal_datatype_pack.h,35)
9000000 9000000 0.638978 hypre_StructAxy (smg2000: struct_axpy.c,25)
8250000 8250000 0.585729 hypre_SMGAxy (smg2000: smg_axpy.c,27)
```

```
8250000 8250000 0.585729 __memcpy_sse3_back (libc-2.17.so)
6750000 6750000 0.479233 hypre_SMG2BuildRAPSym (smg2000: smg2_setup_rap.c,156)
```

This is the CLI view for the hwctime experiment showing the performance information based on the loops in smg2000.

```
openss>>expview -v loops hwctime10
```

```
Exclusive Inclusive % of Total Loop Start Location (Line Number)
PAPI_L1_DCM PAPI_L1_DCM Exclusive
Counts Counts PAPI_L1_DCM
Counts
739500000 765000000 42.813721 smg_residual.c(204)
213000000 256500000 12.331741 cyclic_reduction.c(882)
192000000 227250000 11.115936 cyclic_reduction.c(1022)
41250000 41250000 2.388189 cyclic_reduction.c(851)
41250000 41250000 2.388189 cyclic_reduction.c(835)
40500000 40500000 2.344768 cyclic_reduction.c(851)
39000000 39000000 2.257924 cyclic_reduction.c(851)
24000000 24750000 1.389492 semi_restrict.c(198)
20250000 20250000 1.172384 semi_interp.c(292)
20250000 20250000 1.172384 semi_interp.c(292)
```

This is the CLI view for the hwctime experiment showing the performance information based on the statements in smg2000. Statement 289 in smg2000 had the most level 1 data cache misses during this particular experiment.

```
openss>>expview -vstatements hwctime10
```

```
Exclusive Inclusive % of Total Statement Location (Line Number)
PAPI_L1_DCM PAPI_L1_DCM Exclusive
Counts Counts PAPI_L1_DCM
Counts
573750000 573750000 42.429285 smg_residual.c(289)
111750000 111750000 8.264004 cyclic_reduction.c(1130)
84750000 84750000 6.267332 cyclic_reduction.c(910)
63000000 63000000 4.658902 smg_residual.c(287)
48750000 48750000 3.605103 cyclic_reduction.c(999)
45750000 45750000 3.383250 smg_residual.c(238)
31500000 31500000 2.329451 smg_residual.c(291)
27000000 27000000 1.996672 cyclic_reduction.c(853)
26250000 26250000 1.941209 cyclic_reduction.c(1061)
24000000 24000000 1.774820 cyclic_reduction.c(998)
```

This is the CLI view for the hwctime experiment showing the performance information based on the libraries or linked objects in smg2000. The smg2000 executable had 92% of the level 1 data cache misses during this particular experiment.

```
openss>>expview -v linkedobjects
```

```
Exclusive Inclusive % of Total LinkedObject
PAPI_L1_DCM PAPI_L1_DCM Exclusive
Counts Counts PAPI_L1_DCM
```

Counts			
1297500000	1410000000	92.021277	smg2000
58500000	81750000	4.148936	libopen-pal.so.13.0.2
34500000	101250000	2.446809	libmpi.so.12.0.2
19500000	1410000000	1.382979	libc-2.17.so

### 4.3 Hardware Counter Sampling (hwcsamp) Experiment

The osshwcsamp experiment supports both derived and non-derived PAPI presets and is able to sample up to six counters at one time. Again you can check the available counters by running osshwcsamp with no arguments. All native events are available including architecture specific events listed in the PAPI documentation. Native events are also reported by papi\_native\_avail.

The hardware counter sampling experiment uses a sampling rate (instead of the threshold used in the previous experiments). But like the threshold, the sampling rate is depended on the application and must be balanced between overhead and accuracy. In this case the lower the sampling rate the less samples recorded.

The convenience script for this is experiment is:

```
> osshwcsamp "mpirun -np 256 smg2000 -n 50 50 50" <event_list> <sampling_rate>
```

Note if a counter does not appear in the output, there may be a conflict in the hardware counters. To find conflicts use

```
> papi_event_chooser PRESET <list_of_events>
```

Here is a list of some possible hardware counter combinations to use (list provided by Koushik Ghosh, LLNL).

<b>For Xeon processors:</b>	
PAPI_FP_INS, PAPI_LD_INS, PAPI_SR_INS	Load store info, memory bandwidth needs
PAPI_L1_DCM, PAPI_L1_TCA	L1 cache hit/miss ratios
PAPI_L2_DCM, PAPI_L2_TCA	L2 cache hit/miss ratios
LAST_LEVEL_CACHE_MISSES, LAST_LEVEL_CACHE_REFERENCES	L3 cache info
MEM_UNCORE_RETIRED:REMOTE_DRAM, MEM_UNCORE_RETIRED:LOCAL_DRAM	Local/nonlocal memory access
<b>For Opteron processors:</b>	
PAPI_FAD_INS, PAPI_FML_INS	Floating point add multiply
PAPI_FDV_INS, PAPI_FSQ_INS	Square root and divisions
PAPI_DP_OPS, PAPI_VEC_INS	Floating point and vector instructions
READ_REQUEST_TO_L3_CACHE:ALL_CORES, L3_CACHE_MISSES:ALL_CORES	L3 cache

When selecting PAPI events you must determine if they are a valid combination. In general combination that are valid will pass the test:

```
> papi_event_chooser PRESET event1 event2 ... eventN
```

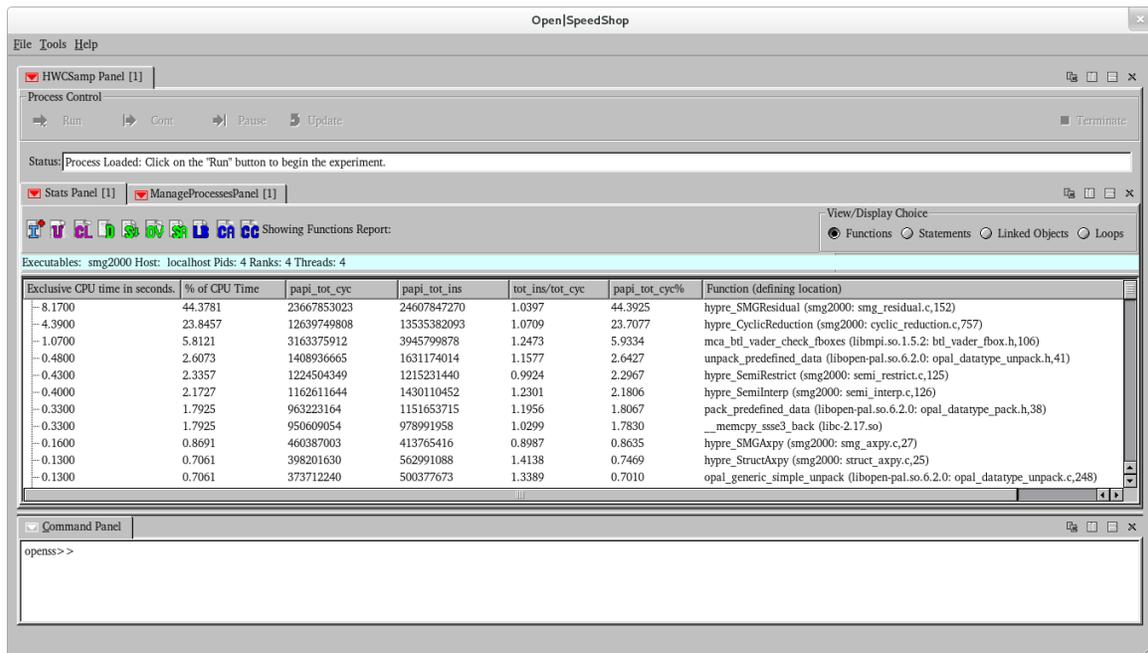
The output for a valid combination will contain:

```
event_chooser.c PASSED
```

Here is an example using PAPI to check if a three-event combination is valid.

```
> papi_event_chooser PRESET PAPI_FP_INS PAPI_LD_INS PAPI_SR_INS
-----PAPI Version :4.1.2.1
Vendor string and code : GenuineIntel (1)
Model string and code : Intel Nehalem (21)
CPU Revision : 5.000000
...
...
PAPI_VEC_SP 0x80000069 No Single precision vector/SIMD instructions
PAPI_VEC_DP 0x8000006a No Double precision vector/SIMD instructions
-----Total events reported: 44
event_chooser.c PASSED
```

Below shows the output of the osshwcsamp experiment with the counters for Total Cycles and Floating Point Operations.



Remember that you do not always need to use the OJSS GUI to examine the output of experiments, you can also use the command line interface to view all of the same information. For example, the same output from above can be seen on the command line:

```
> openss -cli -f smg2000-hwcsamp.openss
openss>>[openss]: The restored experiment identifier is: -x 1
```

```

openss>>expview
Exclusive % of CPU papi_tot_cyc papi_tot_ins tot_ins/tot_cyc papi_tot_cyc% Function (defining location)
CPU time Time
8.1700 44.3781 23667853023 24607847270 1.0397 44.3925 hypre_SMGResidual (smg2000: smg_residual.c,152)
4.3900 23.8457 12639749808 13535382093 1.0709 23.7077 hypre_CyclicReduction (smg2000:
cyclic_reduction.c,757)
1.0700 5.8121 3163375912 3945799878 1.2473 5.9334 mca_btl_vader_check_fboxes (libmpi.so.1.5.2:
btl_vader_fbox.h,106)
0.4800 2.6073 1408936665 1631174014 1.1577 2.6427 unpack_predefined_data (libopen-pal.so.6.2.0:
opal_datatype_unpack.h,41)
0.4300 2.3357 1224504349 1215231440 0.9924 2.2967 hypre_SemiRestrict (smg2000: semi_restrict.c,125)
0.4000 2.1727 1162611644 1430110452 1.2301 2.1806 hypre_SemiInterp (smg2000: semi_interp.c,126)

openss>>expview -v linkedobjects

Exclusive % of CPU papi_tot_cyc papi_tot_ins tot_ins/tot_cyc papi_tot_cyc% LinkedObject
CPU time Time
14.4400 78.3931 41735382047 44436967182 1.0647 78.2394 smg2000
1.9400 10.5320 5687487004 7106186325 1.2494 10.6621 libmpi.so.1.5.2
1.3400 7.2747 3918861297 4788179789 1.2218 7.3465 libopen-pal.so.6.2.0
0.6700 3.6374 1918417268 2138815631 1.1149 3.5964 libc-2.17.so
0.0300 0.1629 83014429 77541115 0.9341 0.1556 libpthread-2.17.so
18.4200 100.0000 53343162045 58547690042 1.0976 100.0000 Report Summary

```

### 4.3.1 Hardware Counter Sampling (hwcsamp) experiment performance data gathering

The hardware counter sampling experiment convenience script is “osshwcsamp”. Use this convenience script in this manner to gather counter values for unique up to six (6) hardware counters:

```
osshwcsamp “how you normally run your application” <papi event list> < sampling rate>
```

#### 4.3.1.1 Hardware Counter Sampling (hwcsamp) experiment parameters

The hwcsamp experiment is timer based not threshold based. What that means is a timer is used to periodically interrupt the processor. For the hwcsamp experiment, each time the timer interrupts the processor; the values of the hardware counter events specified will be read up and reset to 0 for the next timer cycle. This is repeated until the program finishes. O|SS allows the user to control the sampling rate.

The following is an example of how to gather data for the smg2000 application on a Linux cluster platform using the osshwcsamp convenience script and specifying a specific set of PAPI hwc events. In the next example the user is choosing to only sample 45 times a second instead of the default 100 times a second. Why would you want to do this? One reason would be to save database size, a lower sampling rate may give an accurate portrayal of the application behavior.

```
> osshwcsamp “mpirun -np 256 smg2000 -n 50 50 50” PAPI_L1_DCM,PAPI_L2_DCA,PAPI_L2_DCM,PAPI_L3_DCA,PAPI_L3_TCM
```

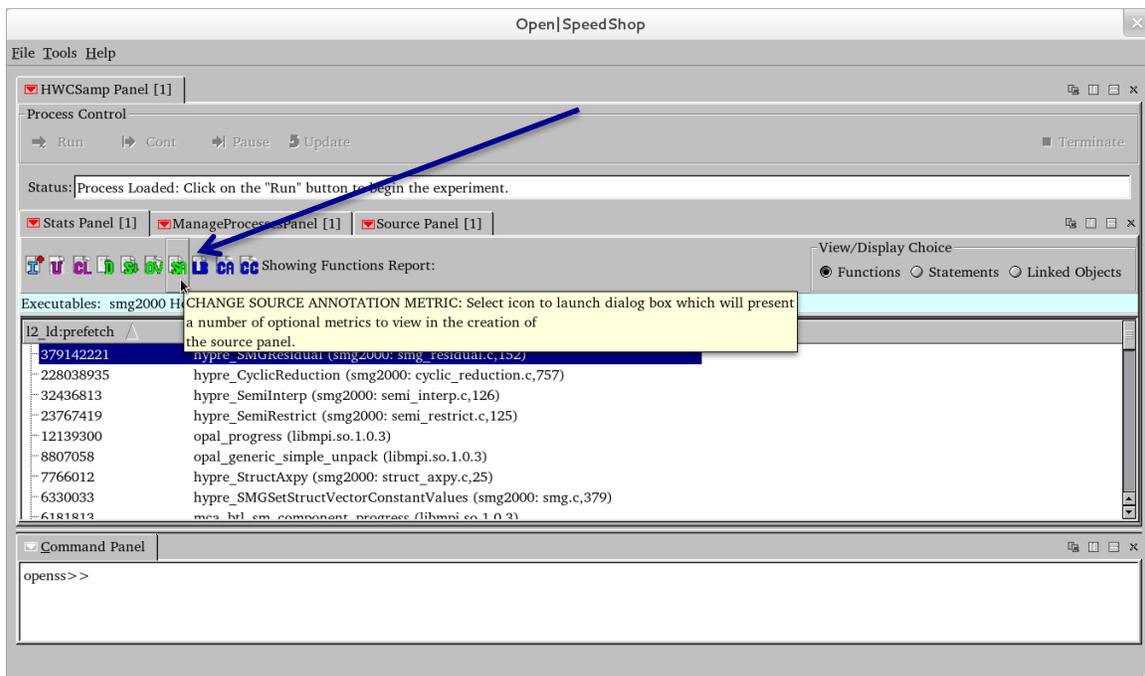
```
> osshwcsamp “mpirun -np 256 smg2000 -n 50 50 50” PAPI_L1_DCM,PAPI_L2_DCA,PAPI_L2_DCM 45
```

### 4.3.2 Hardware Counter Sampling (hwcsamp) experiment performance data viewing with GUI

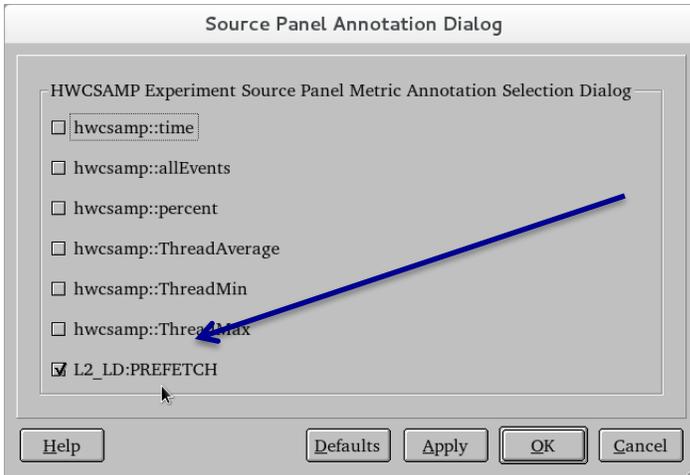
To launch the GUI on any experiment, use “openss -f <database name>”.

#### 4.3.2.1 Getting the PAPI counter as the GUIs Source Annotation Metric

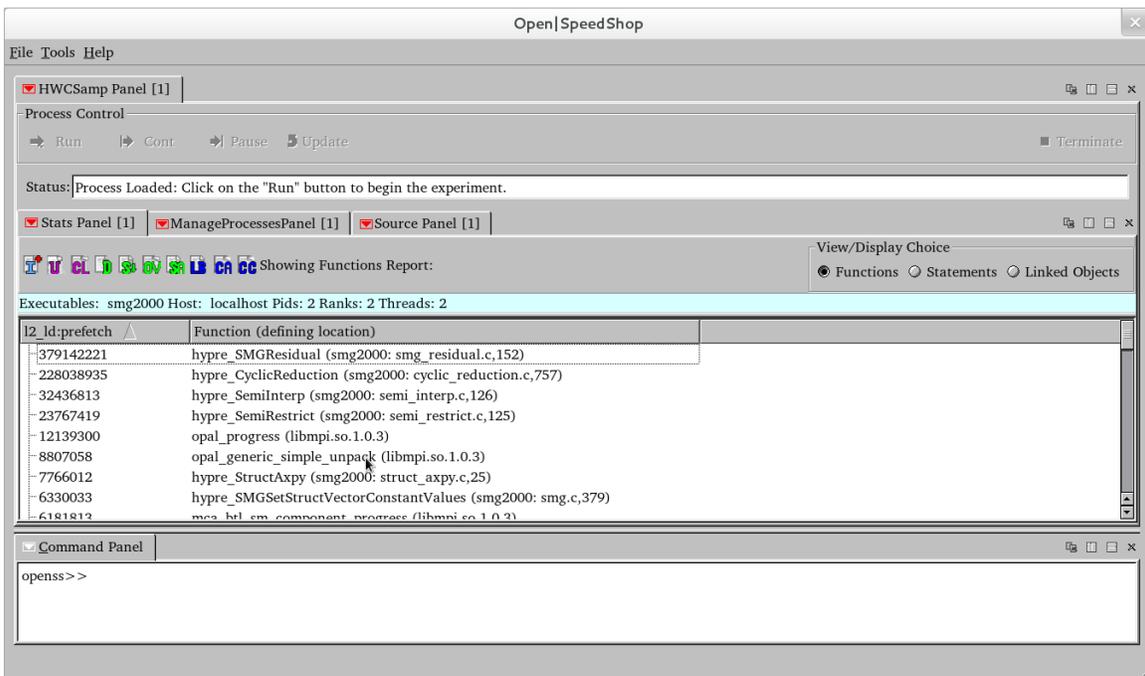
In order to make one of the PAPI or native hardware counters the counter that will show up in the source view, one can click on the “SA” icon, which represents Source Annotation. This brings up an option dialogue that allows you to choose the source annotation metric.



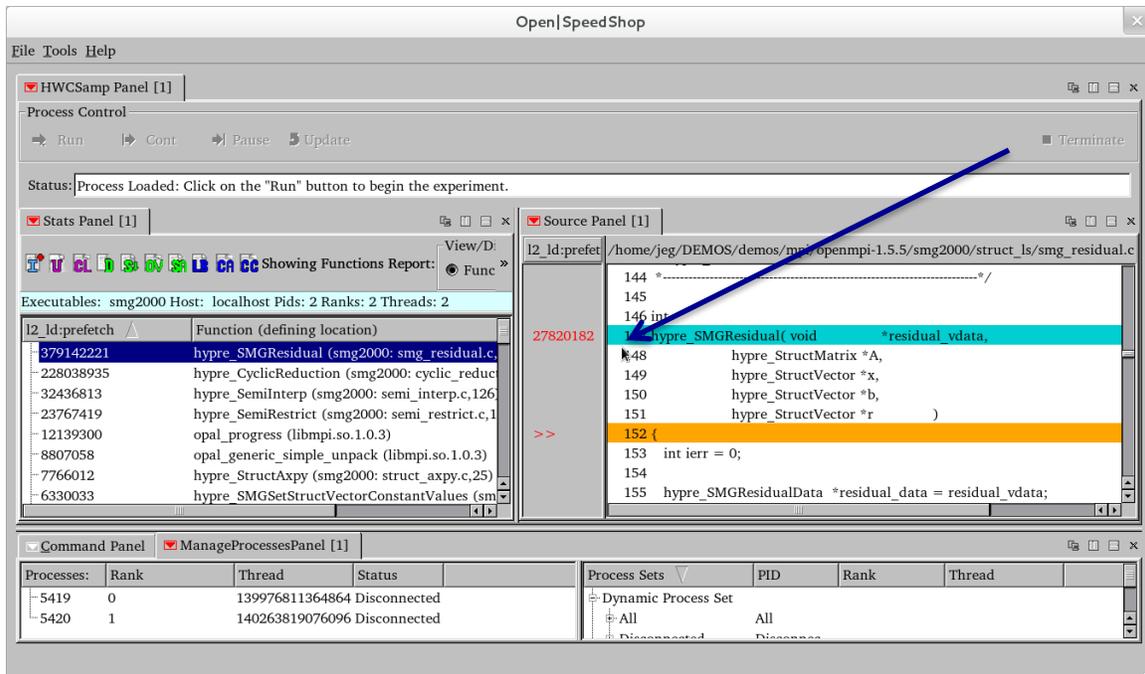
In this example the native counter we want to choose is L2\_LD\_PREFETCH. When we click to choose that counter and click on “OK” the Stats Panel view will regenerate and the source annotation metric will become L2\_LD\_PREFETCH.



The regenerated view now shows the results for only L2\_LD:PREFETCH.



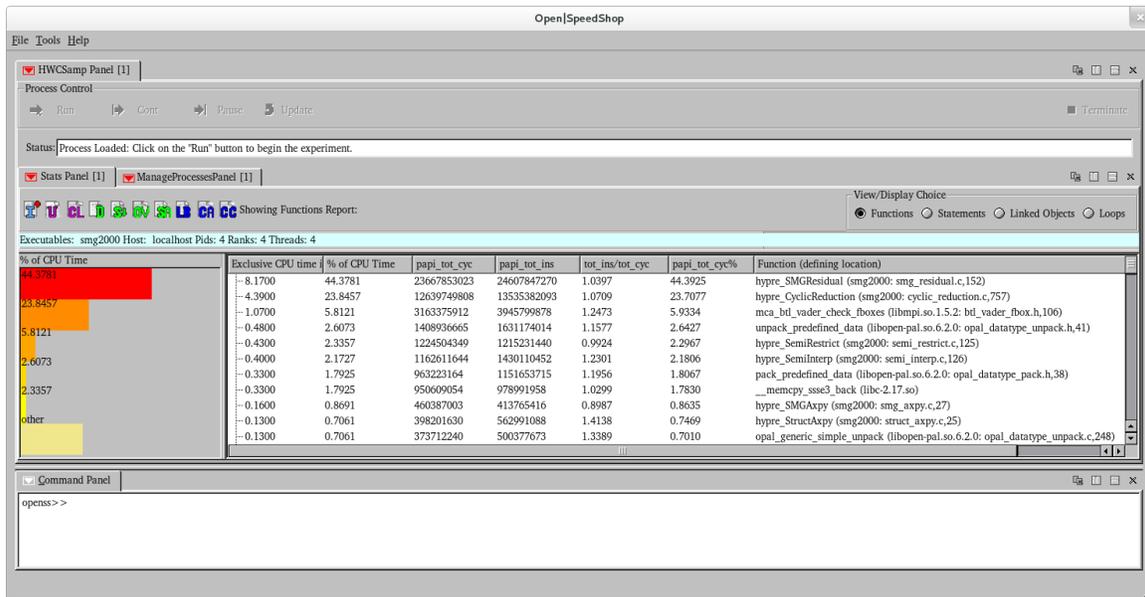
Now double clicking on the Stats Panel result line of choice will focus the source panel and use the PAPI or native counter that was chosen by using the Source Annotation dialog.



#### 4.3.2.2 Viewing Hardware Counter Sampling Data with the GUI

To launch the GUI on any experiment, use “opens -f <database name>”.

The GUI view below represents an example of the default view for the hardware counter sampling (hwcsamp) experiment. In the default view the first set of performance data shown is program counter exclusive time (where the program is statistically spending its time) and the percentage of time spent in each function of the program. The next information is the hardware counter event counts listed in columns by the hardware counter event. Column three represents the counts that were recorded for PAPI\_TOT\_CYC and column four represents the counts for PAPI\_TOT\_INS. What this view can indicate to the viewer is whether or not the specified hardware counter events are occurring and if they are, then how prevalent are they. With this information the user could isolate down to see exactly where a particular event is occurring by using the hwc or hwctime experiment. These two experiments are threshold based. Which ultimately means you can map the performance data back to the source because the actual event triggered the recording of the counts of the event. This hwcsamp experiment is timer based, so O|SS cannot take you to the line of source exactly where the hardware counter events are happening. hwcsamp is an overview experiment that tells the user which events are occurring and if they are occurring in numbers that would warrant using the hwc or hwctime experiment to pinpoint where in the source the specified hardware counter event is actually occurring.



### 4.3.3 Hardware Counter Sampling (hwcsamp) experiment CLI performance data viewing

To launch the CLI on any experiment, use “opens -cli -f <database name>“. The following example was run on the Yellowstone platform at NCAR/UCAR using the job script shown below.

#### 4.3.3.1 Job Script and osshwcsamp command

```
#!/bin/csh
#
# LSF batch script to run an MPI application
#
#BSUB -P Pnnnnnnnn # project code
#BSUB -W 00:30 # wall-clock time (hrs:mins)
#BSUB -n 64 # number of tasks in job
#BSUB -R "span[ptile=4]" # run 4 MPI tasks per node
#BSUB -j sweep3d-hwcsamp # job name
#BSUB -o sweep3d-hwcsamp.%J.out # output file name in which %J is replaced by the job ID
#BSUB -e sweep3d-hwcsamp.%J.err # error file name in which %J is replaced by the job ID
#BSUB -q regular # queue

module load openspeedshop

mkdir -p /glade/scratch/${USER}/sweep3d
rm -rf /glade/scratch/${USER}/sweep3d/hwcsamp
mkdir /glade/scratch/${USER}/sweep3d/hwcsamp
setenv OPENS_RAWDATA_DIR /glade/scratch/${USER}/sweep3d/hwcsamp

setenv REQUEST_SUSPEND_HPC_STAT 1

echo "running (on compute node): osshwcsamp"
osshwcsamp "mpirun.lsf /glade/u/home/galaro/demos/sweep3d/orig/sweep3d.mpi"
PAPI_L1_DCM,PAPI_L1_ICM,PAPI_L1_TCM,PAPI_L1_LDM,PAPI_L1_STM
```

### 4.3.3.2 osshwcsamp experiment default CLI view

The table below describes information that is included in the hwcsamp experiment default view when no alternative PAPI hardware counter arguments are specified for the osshwcsamp experiment.

Column Name	Column Definition
Exclusive CPU Time	Aggregated total exclusive time spent in the application function corresponding to this row of data.
% of CPU Time	Percentage of exclusive time spent in the function corresponding to this row of data relative to the total application exclusive time for all the application functions.
PAPI_TOT_CYC	Number of hardware events corresponding to the hardware independent PAPI_TOT_CYC PAPI event. This value is based on reading the hardware counter event buffers based on sampling. This means this data may not accurately reflect where in the source these events occurred. It is an approximation of what is going in the application, but not a mapping back to the source lines. Use the hwc and hwctime experiments for that.
PAPI_TOT_INS	Number of hardware events corresponding to the hardware independent PAPI_TOT_INS PAPI event. This value is based on reading the hardware counter event buffers based on sampling. This means this data may not accurately reflect where in the source these events occurred. It is an approximation of what is going in the application, but not a mapping back to the source lines. Use the hwc and hwctime experiments for that.
TOT_INS/TOT_CYC	This is the graduated instructions per cycle, which is the ratio between the approximation of the total number of instructions divided by the total number of cycles
% of TOT_CYC	The percentage of PAPI_TOT_CYC events for this function relative to the number of PAPI_TOT_CYC events that occurred in all the application functions.

This is a default CLI view for the hwcsamp experiment:

```
Exclusive % of CPU papi_tot_cyc papi_tot_ins tot_ins/tot_cyc papi_tot_cyc% Function (defining location)
CPU time Time
in
seconds.
74.0600 99.8786 177712237021 51989184616 0.2925 99.8787 main (nbody: nbody-mpi.c,71)
0.0400 0.0539 95958566 28058948 0.2924 0.0539 fesetenv (libm-2.19.so)
0.0300 0.0405 71987793 21053819 0.2925 0.0405 __sqrt_finite (libm-2.19.so)
0.0100 0.0135 23864331 6996727 0.2932 0.0134 memcpy (libc-2.19.so)
0.0100 0.0135 23995616 7018006 0.2925 0.0135 fegetround (libm-2.19.so)
74.1500 100.0000 177928043327 52052312116 0.2925 100.0000 Report Summary
```

This is the output from an osshwcsamp experiment non-default experiment where PAPI\_L1\_DCM,PAPI\_L1\_ICM,PAPI\_L1\_TCM,PAPI\_L1\_LDM,PAPI\_L1\_STM were specified on the osshwcsamp command.

```
openss -cli -f L1-64PE-sweep3d.mpi-hwcsamp.openss
```

```

openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview -v summary

Exclusive  % of CPU papi_l1_dcm papi_l1_icm papi_l1_tcm papi_l1_ldm papi_l1_stm Function (defining location)
CPU time in Time
seconds.
824.870000 38.689781 8646497071 117738843 8764235914 8396159476 196649065 __libc_poll (libc-2.12.so)
799.300000 37.490443 46691996441 367096209 47059092650 46247555479 281624221 sweep (sweep3d.mpi:
sweep.f,2)
75.000000 3.517807 782716992 10680760 793397752 757322217 20159725
PAMI::Interface::Context<PAMI::Context>::advance (libpami.so: ContextInterface.h,158)
55.750000 2.614903 597583047 8038242 605621289
579127274 14647999 LapiImpl::Context::Advance<true, true, false> (libpami.so: Context.h,220)
52.970000 2.484510 550761926 7569975 558331901 535841812 11563657 __libc_enable_asynccancel (libc-
2.12.so)
49.850000 2.338169 518605433 6979361 525584794 502551336 12757207 _lapi_dispatcher<false> (libpami.so:
lapi_dispatcher.c,57)
48.080000 2.255149 488545916 6784192 495330108 476065093 9649598 LapiImpl::Context::TryLock<true, true,
false> (libpami.so: Context.h,198)
47.750000 2.239671 479947719 6732551 486680270 471343480 6436257 __libc_disable_asynccancel (libc-
2.12.so)
26.680000 1.251401 275998769 3888499 279887268 269841454 4697170 udp_read_callback (libpamiudp.so:
lapi_udp.c,538)
25.880000 1.213878 1522697263 12118336 1534815599 1507685061 9619348 __intel_ssse3_rep_memcpy
(libirc.so)
21.960000 1.030014 223197680 3086626 226284306 215787794 5879517 _lapi_shm_dispatcher (libpami.so:
lapi_shm.c,2283)
14.910000 0.699340 154744623 2075688 156820311 149803306 3979337 LapiImpl::Context::CheckContext
(libpami.so: CheckParam.cpp,21)
13.990000 0.656188 151052863 2000330 153053193 146967548 3167039 LapiImpl::Context::Unlock<true, true,
false> (libpami.so: Context.h,204)

```

### 4.3.3.2 osshwcsamp experiment Status command and CLI view

```

openss>>expstatus

Experiment definition
{ # ExpId is 1, Status is NonExistent, Saved database is L1-64PE-sweep3d.mpi-hwcsamp.openss
Performance data spans 1:7.958138 mm:ss from 2013/03/27 22:32:45 to 2013/03/27 22:33:53
Executables Involved:
  sweep3d.mpi
Currently Specified Components:
-h ys6128 -p 2765 -t 47176895393312 -r 3 (sweep3d.mpi)
-h ys6128 -p 2766 -t 47824321252896 -r 0 (sweep3d.mpi)
-h ys6128 -p 2767 -t 47369830317600 -r 1 (sweep3d.mpi)
-h ys6128 -p 2768 -t 47378742910496 -r 2 (sweep3d.mpi)
-h ys6129 -p 22862 -t 47327259860512 -r 5 (sweep3d.mpi)
-h ys6129 -p 22863 -t 47201888194080 -r 6 (sweep3d.mpi)
-h ys6129 -p 22864 -t 47185544437280 -r 7 (sweep3d.mpi)
...
-h ys6250 -p 11462 -t 47028080107040 -r 63 (sweep3d.mpi)
-h ys6250 -p 11463 -t 47600632852000 -r 60 (sweep3d.mpi)
-h ys6250 -p 11464 -t 47494028697120 -r 61 (sweep3d.mpi)
-h ys6250 -p 11465 -t 47944527175200 -r 62 (sweep3d.mpi)
Previously Used Data Collectors:
  hwcsamp
  Metrics:
  hwcsamp::exclusive_detail
  hwcsamp::percent
  hwcsamp::threadAverage
  hwcsamp::threadMax
  hwcsamp::threadMin
  hwcsamp::time
  Parameter Values:
  hwcsamp::event = PAPI_L1_DCM,PAPI_L1_ICM,PAPI_L1_TCM,PAPI_L1_LDM,PAPI_L1_STM
  hwcsamp::sampling_rate = 100
  Available Views:

```

```
hwcsamp
}
```

### 4.3.3.3 osshwsamp experiment Load Balance command and CLI view

```
openss>>expview -m loadbalance
```

```
Max CPU Rank  Min CPU Rank  Average Function (defining location)
Time of Time of CPU Time
Across Max Across Min Across
Ranks(s) Ranks(s) Ranks(s)
14.890000 28 10.950000 27 12.888594 __libc_poll (libc-2.12.so)
14.270000 47 11.780000 51 12.489062 sweep (sweep3d.mpi: sweep.f,2)
1.620000 43 0.840000 37 1.171875 PAMI::Interface::Context<PAMI::Context>::advance (libpami.so:
ContextInterface.h,158)
1.320000 16 0.570000 3 0.871094 LapiImpl::Context::Advance<true, true, false> (libpami.so: Context.h,220)
1.130000 60 0.500000 2 0.778906 _lapi_dispatcher<false> (libpami.so: lapi_dispatcher.c,57)
1.110000 35 0.520000 49 0.751250 LapiImpl::Context::TryLock<true, true, false> (libpami.so: Context.h,198)
1.030000 42 0.600000 12 0.827656 __libc_enable_asynccancel (libc-2.12.so)
0.950000 62 0.520000 38 0.746094 __libc_disable_asynccancel (libc-2.12.so)
0.700000 6 0.200000 59 0.343125 _lapi_shm_dispatcher (libpami.so: lapi_shm.c,2283)
0.630000 33 0.250000 0 0.404375 __intel_ssse3_rep_memcpy (libirc.so)
0.600000 18 0.270000 16 0.416875 udp_read_callback (libpamiudp.so:
```

### 4.3.3.4 osshwsamp experiment Linked Object command and CLI view

```
openss>>expview -v linkedobjects
```

```
Exclusive % of CPU papi_l1_dcm papi_l1_icm papi_l1_tcm papi_l1_ldm papi_l1_stm LinkedObject
CPU time in Time
seconds.
928.310000 43.541541 9818946796 133244862 9952191658 9543597734 215608918 libc-2.12.so
811.920000 38.082373 47212355914 369525459 47581881373 46596204924 441601622 sweep3d.mpi
311.490000 14.610157 3356646038 44875637 3401521675 3255300343 80090932 libpami.so
29.640000 1.390237 1824778610 12931604 1837710214 1680978945 127174346 libirc.so
26.930000 1.263127 287313329 3994016 291307345 281053971 4763152 libpamiudp.so
22.250000 1.043616 1049603690 9037920 1058641610 1033650896 11422120 libpthread-2.12.so
1.440000 0.067542 72649683 620083 73269766 71327993 1007704 libmpich.so.3.3
0.020000 0.000938 1286256 23770 1310026 1232178 5222 ld-2.12.so
0.010000 0.000469 327 394 721 313 13 librt-2.12.so
2132.010000 100.000000 63623580643 574253745 64197834388 62463347297 881674029 Report Summary
openss>>
```

### 4.3.3.5 osshwsamp experiment only the hwcsamp PAPI events CLI view

The -m allEvents option prints only the PAPI event values and not the program counter sampling exclusive time and percentage values.

```
openss -cli -f L1-64PE-sweep3d.mpi-hwcsamp.openss
```

```
openss>>[openss]: The restored experiment identifier is: -x 1
```

```
openss>>expview -m allEvents
```

```
papi_l1_dcm papi_l1_icm papi_l1_tcm papi_l1_ldm papi_l1_stm Function (defining location)
8646497071 117738843 8764235914 8396159476 196649065 __libc_poll (libc-2.12.so)
46691996441 367096209 47059092650 46247555479 281624221 sweep (sweep3d.mpi: sweep.f,2)
782716992 10680760 793397752 757322217 20159725 PAMI::Interface::Context<PAMI::Context>::advance
597583047 8038242 605621289 579127274 14647999 LapiImpl::Context::Advance<true, true, false>
550761926 7569975 558331901 535841812 11563657 __libc_enable_asynccancel (libc-2.12.so)
518605433 6979361 525584794 502551336 12757207 _lapi_dispatcher<false> (libpami.so:
lapi_dispatcher.c,57)
488545916 6784192 495330108 476065093 9649598 LapiImpl::Context::TryLock<true, true, false>
479947719 6732551 486680270 471343480 6436257 __libc_disable_asynccancel (libc-2.12.so)
275998769 3888499 279887268 269841454 4697170 udp_read_callback (libpamiudp.so: lapi_udp.c,538)
1522697263 12118336 1534815599 1507685061 9619348 __intel_ssse3_rep_memcpy (libirc.so)
223197680 3086626 226284306 215787794 5879517 _lapi_shm_dispatcher (libpami.so: lapi_shm.c,2283)
```

```
154744623 2075688 156820311 149803306 3979337 LapiImpl::Context::CheckContext (libpami.so:
CheckParam.cpp,21)
151052863 2000330 153053193 146967548 3167039 LapiImpl::Context::Unlock<true, true, false>
(libpami.so: Context.h,204)
```

## 5 I/O Tracing and I/O Profiling

### 5.1 Open|SpeedShop I/O Tracing General Usage

The Open|SpeedShop (O|SS) *io* and *iot* I/O function tracing experiments wrap the most common I/O functions, record the time spent in each I/O function, record the call path along which I/O function was called, record the time spent along each call path to an I/O function, and record the number of times each function was called. In addition, the *iot* experiment also records information about each individual I/O function call. The values of the arguments and the return value from the I/O function are recorded.

### 5.2 I/O Base Tracing (*io*) experiment

The base I/O tracing experiment gathers data for the following I/O functions: close, creat, creat64, dup, dup2, lseek, lseek64, open, open64, pipe, pread, pread64, pwrite, pwrite64, read, readv, write, and writev. It is a trace type experiment that wraps the real I/O calls and records information before and after calling the real I/O functions. This, base, I/O experiment records the basic I/O information as stated in the introductory section, but does not record the arguments to each call. That is done in the extended (*iot*) experiment.

#### 5.2.1 I/O Base Tracing (*io*) experiment performance data gathering

The base I/O tracing (*io*) experiment convenience script is “ossio”. Use this convenience script in this manner to gather base I/O tracing performance data:

```
ossio “how you normally run your application” <list of I/O function(s)>
```

The following is an example of how to gather data for the IOP application on a Linux cluster platform using the ossio convenience script. It gathers performance data for all the I/O functions because there are no list I/O functions specified after the quoted application run command.

```
ossio "srun -n 512 ./IOR"
```

#### 5.2.2 I/O Base Tracing (*io*) experiment performance data viewing with CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”.

#### 5.2.3 I/O Base Tracing (*io*) experiment performance data viewing with GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

### 5.3 I/O Extended Tracing (iot) experiment

#### 5.3.1 I/O Extended Tracing (iot) experiment performance data gathering

The extended I/O tracing (iot) experiment convenience script is “ossiot”. Use this convenience script in this manner to gather extended I/O tracing performance data:

```
ossiot “how you normally run your application” <list of I/O function(s)>
```

The following is an example of how to gather data for the IOP application on a Linux cluster platform using the ossiot convenience script. It gathers performance data for all the I/O functions because there are no list I/O functions specified after the quoted application run command.

```
ossiot "srun -n 512 ./IOR"
```

#### 5.3.2 I/O Extended Tracing (iot) experiment performance data viewing with GUI

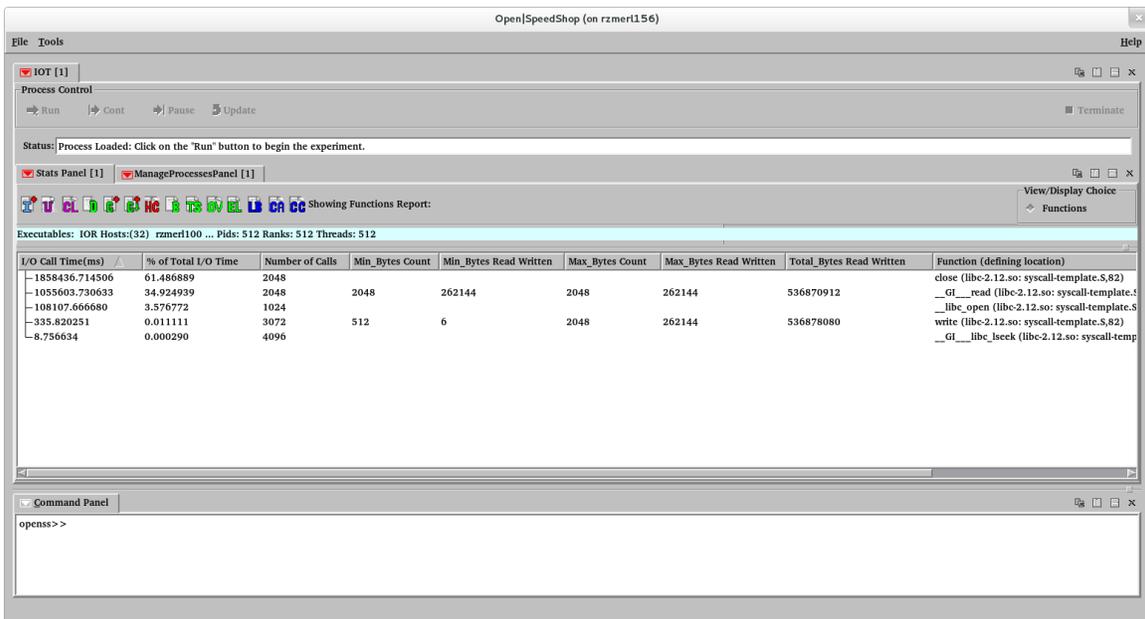
To launch the GUI on any experiment, use “openss -f <database name>”.

This is the default GUI view for the iot experiment. This view gives a summary of the I/O functions that were called, how many times they were called and the amount of time spent in each function. The percentage of the total I/O time is also attributed to each I/O function. The time is aggregated (totaled) across all the threads, ranks, or processes that were part of the application. The table below describes what the columns of data represent.

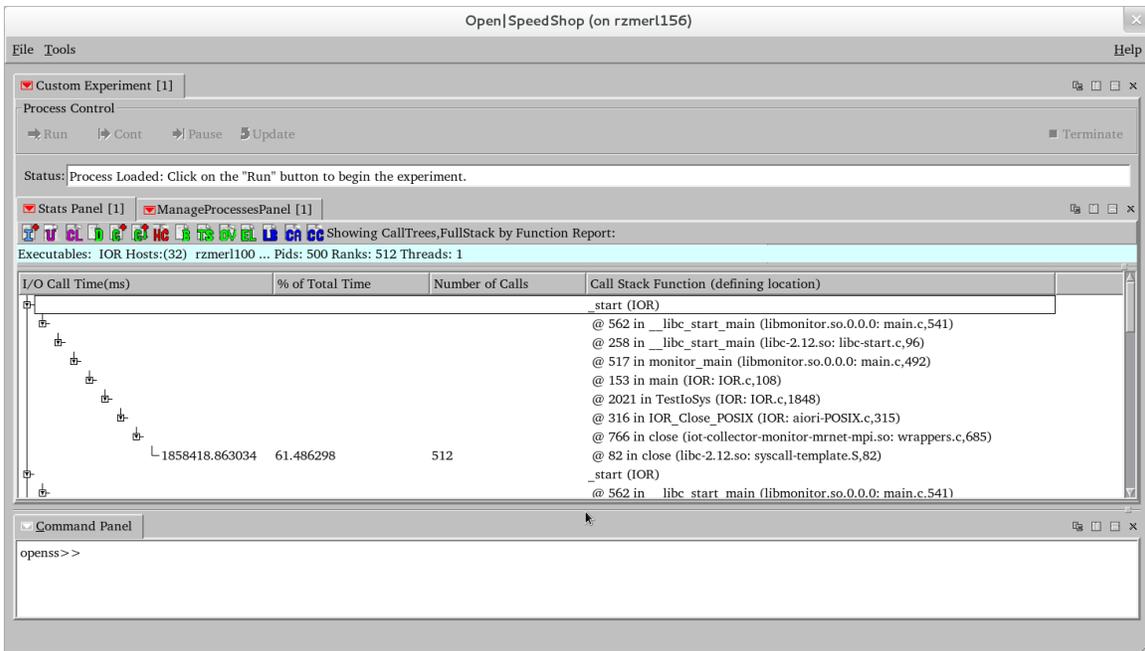
The functions that called the I/O functions are available by choosing one of the call path views.

Column Name	Column Definition
I/O Call Time	Aggregated total exclusive time spent in the I/O function corresponding to this row of data.
% of I/O Total Time	Percentage of exclusive time relative to the total time spent in the I/O function corresponding to this row of data.
Number of Calls	Total number of calls to the I/O function corresponding to this row of data.
Min Bytes Count	The number of times minimum bytes read or written by the corresponding I/O function occurred during this experiment.

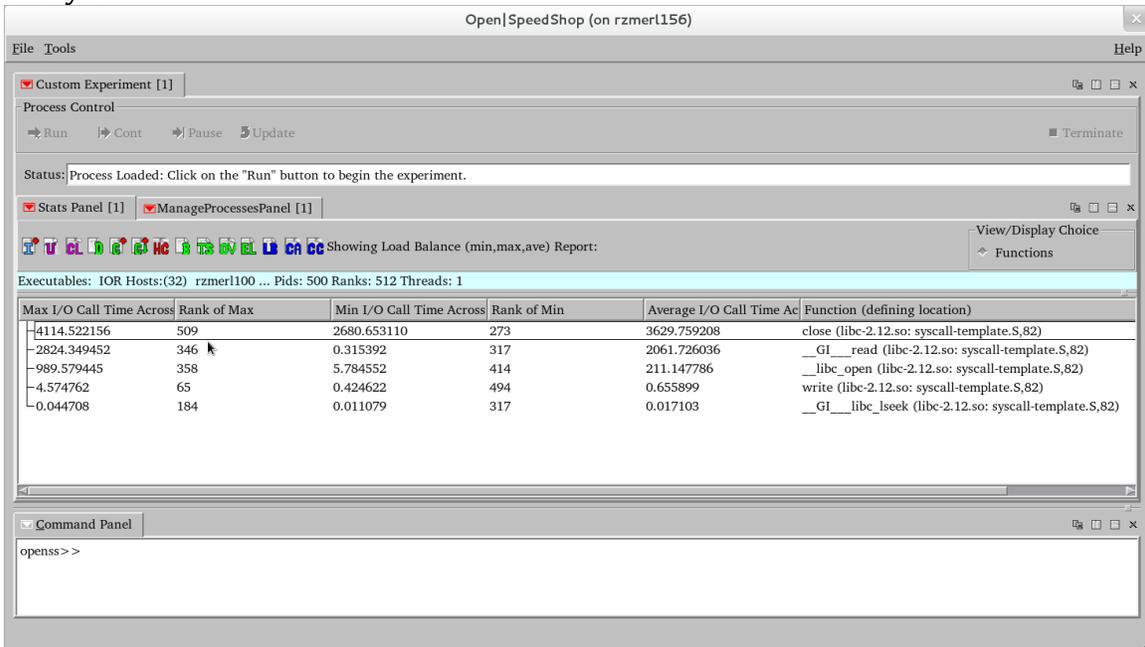
Column Name	Column Definition
Min Bytes Read or Written	The minimum number of bytes that were read or written by the corresponding I/O function.
Max Bytes Count	The number of times maximum bytes read or written by the corresponding I/O function occurred during this experiment.
Max Bytes Read or Written	The maximum number of bytes that were read or written by the corresponding I/O function.
Total Bytes Read or Written	The total number of bytes read or written by the corresponding function. This number only represents the totals for the number of bytes read or written based on the I/O function called.



Here the user has chosen the C+ view icon and the Stats Panel now shows all the call paths in the user's application. This view shows every possible call paths through the source to all the I/O functions that were called during the execution of this application. From this one could validate that this is expected behavior and if not find where the I/O in this application is not behaving as expected.

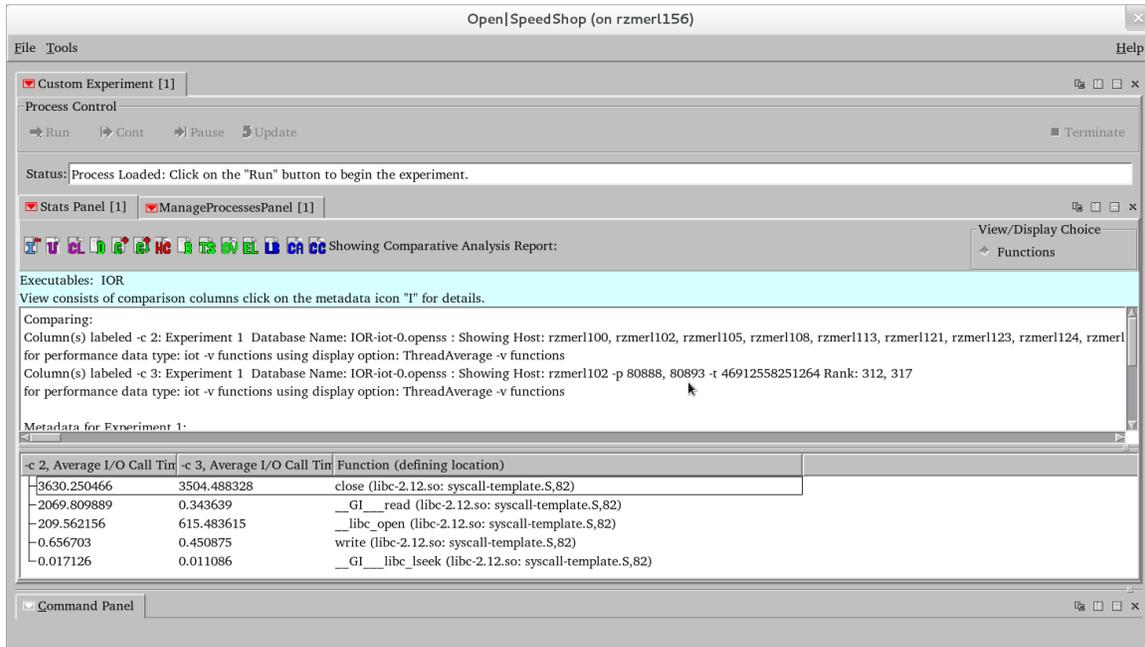


This view is the load balance view, which gives the min, max, average values for the I/O function call time across all the ranks in this application. In this view we are seeing some wide ranges between the min and max values for some of the I/O functions. It may be useful to see if we can identify the ranks by using the Cluster Analysis view.



This view, generated by choosing CA icon, the shows that there are two groups of ranks where the I/O is performing in similar manner. For group 2 (labeled -c 3 below), there are two ranks where the rest of the 512 ranks perform like group 1 (labeled -c 2 below). Investigation by examining ranks 312 or 317 by comparing it

to one of the ranks in the other group could shed some light on why group 2 is not similar to the rest. This may or may not be significant, but is here for illustration.



### 5.3.3 I/O Extended Tracing (iot) experiment performance data viewing with CLI

To launch the CLI on any experiment, use “`opends -cli -f <database name>`”.

The command line interface (CLI) can provide the same data options as the graphical user interface (GUI) views. Here are some examples of the performance data that can be viewed and the commands in order to generate the CLI views. The following table describes header and meaning of the default iot view CLI columns.

Column Name	Column Definition
I/O Call Time	Aggregated total exclusive time spent in the I/O function corresponding to this row of data.
% of I/O Total Time	Percentage of exclusive time relative to the total time spent in the I/O function corresponding to this row of data.
Number of Calls	Total number of calls to the I/O function corresponding to this row of data.
Min Bytes Count	The number of times minimum bytes read or written by the corresponding I/O function occurred during this experiment.
Min Bytes Read or Written	The minimum number of bytes that were read or written by the corresponding I/O function.
Max Bytes Count	The number of times maximum bytes read or written by the corresponding I/O function occurred during this experiment.
Max Bytes Read or Written	The maximum number of bytes that were read or written by the corresponding I/O function.



```

I/O Call   % of Number Call Stack Function (defining location)
Time(ms)  Total of
         Time Calls
      _start (IOR)
      > @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
      >> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
      >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
      >>>> @ 153 in main (IOR: IOR.c,108)
      >>>>> @ 2021 in TestIoSys (IOR: IOR.c,1848)
      >>>>>> @ 316 in IOR_Close_POSIX (IOR: aiori-POSIX.c,315)
      >>>>>>> @ 766 in close (iot-collector-monitor-mrnet-mpi.so: wrappers.c,685)
1858418.863034 61.486298 512 >>>>>>> @ 82 in close (libc-2.12.so: syscall-template.S,82)
      _start (IOR)
      > @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
      >> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
      >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
      >>>> @ 153 in main (IOR: IOR.c,108)
      >>>>> @ 2173 in TestIoSys (IOR: IOR.c,1848)
      >>>>>> @ 2611 in WriteOrRead (IOR: IOR.c,2562)
      >>>>>>> @ 251 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
      >>>>>>>> @ 223 in read (iot-collector-monitor-mrnet-mpi.so: wrappers.c,137)
1055603.730633 34.924939 2048 >>>>>>>> @ 82 in __GI__read (libc-2.12.so: syscall-template.S,82)
      _start (IOR)
      > @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
      >> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
      >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
      >>>> @ 153 in main (IOR: IOR.c,108)
      >>>>> @ 2004 in TestIoSys (IOR: IOR.c,1848)
      >>>>>> @ 104 in IOR_Create_POSIX (IOR: aiori-POSIX.c,74)
      >>>>>>> @ 670 in open64 (iot-collector-monitor-mrnet-mpi.so: wrappers.c,608)
103350.518692 3.419380 512 >>>>>>>> @ 82 in __libc_open (libc-2.12.so: syscall-template.S,82)
      _start (IOR)
      > @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
      >> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
      >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
      >>>> @ 153 in main (IOR: IOR.c,108)
      >>>>> @ 2161 in TestIoSys (IOR: IOR.c,1848)
      >>>>>> @ 195 in IOR_Open_POSIX (IOR: aiori-POSIX.c,173)
      >>>>>>> @ 670 in open64 (iot-collector-monitor-mrnet-mpi.so: wrappers.c,608)
4757.147988 0.157392 512 >>>>>>>> @ 82 in __libc_open (libc-2.12.so: syscall-template.S,82)
      _start (IOR)
      > @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
      >> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
      >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
      >>>> @ 153 in main (IOR: IOR.c,108)
      >>>>> @ 2013 in TestIoSys (IOR: IOR.c,1848)
      >>>>>> @ 2608 in WriteOrRead (IOR: IOR.c,2562)
      >>>>>>> @ 244 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
      >>>>>>>> @ 321 in write (iot-collector-monitor-mrnet-mpi.so: wrappers.c,239)
316.176763 0.010461 2048 >>>>>>>> @ 82 in write (libc-2.12.so: syscall-template.S,82)

```

## 5.3 I/O Lightweight Profiling (iop) General Usage

The Open|SpeedShop (O|SS) *iop* I/O function profiling experiment wraps the most common I/O functions, records the time spent in each I/O function, record the call path along which I/O function was called, record the time spent along each call path to an I/O function, and record the number of times each function was called.

### 5.3.1 I/O Profiling (iop) experiment performance data gathering

The I/O Profiling (iop) experiment convenience script is “ossiop”. Use this convenience script in this manner to gather lightweight I/O profiling performance data:

```
ossiop "how you normally run your application"
```

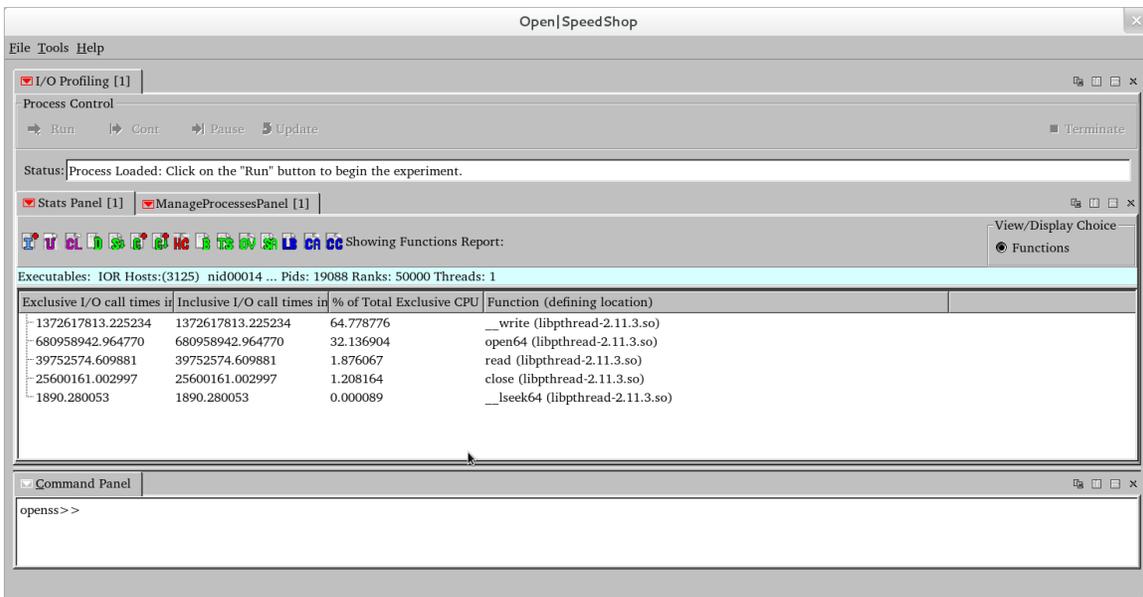
The following is an example of how to gather data for the IOP application on the Cray platform using the ossiop convenience script.

```
ossiop "aprun -n 64 ./IOR"
```

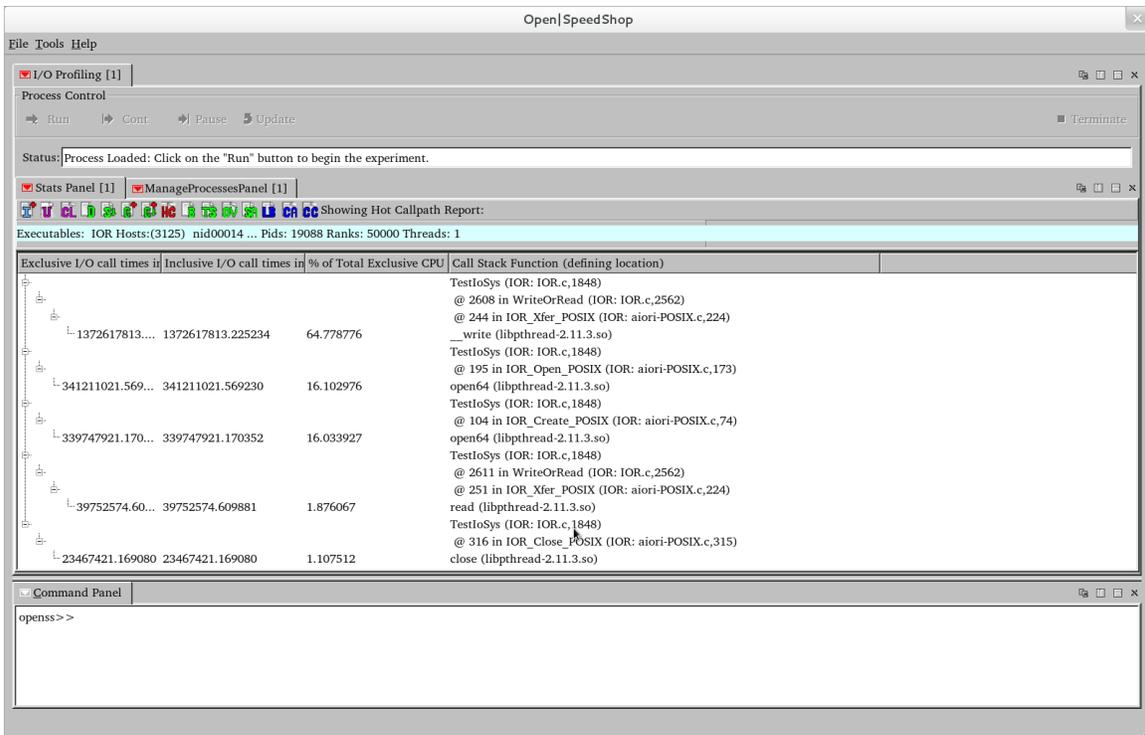
### 5.3.2 I/O Profiling (iop) experiment performance data viewing with GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

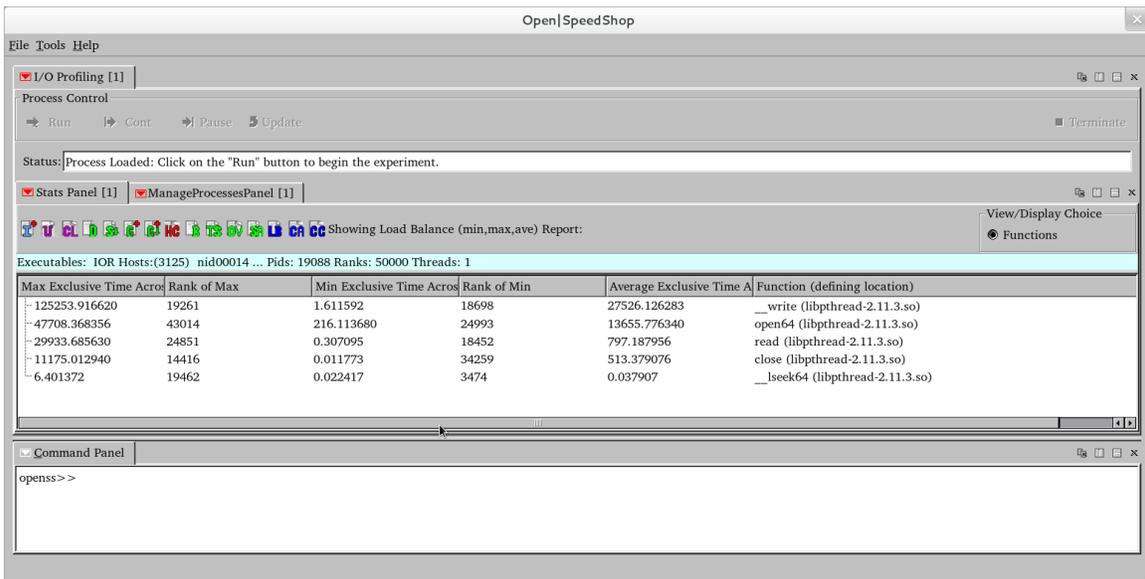
The first image, below, shows the default view for the iop experiment run on a 50000 rank IOR application job. The performance information in the default view is the time spent in I/O functions and the percentage of time spent in each I/O function.



In the image below, the hot call path view for the iop experiment run on a 50000 rank IOR application job is displayed. The performance information in the hot call path view is the top five call paths to each of the I/O functions that took the most time, time spent in I/O functions and the percentage of time spent in each I/O function.



This image shows the min, max, average time spent in each of the I/O functions showing the rank of the minimum value and the rank of the maximum value for each of the I/O functions. This view indicates if there is an imbalance relative to the I/O in the application being run. This may or may not be expected.



### 5.3.3 I/O Profiling (iop) experiment performance data viewing with CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”.

The command line interface (CLI) can provide the same data options as the graphical user interface (GUI) views. Here are some examples of the performance data that can be viewed and the commands in order to generate the CLI views.

```

> openss -cli -f IOR-iop-1.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview

Exclusive Inclusive % of Function (defining location)
I/O call I/O call Total
times in times in Exclusive
seconds. seconds. CPU Time
38297.33 38297.33 96.46 __write (libpthread-2.11.3.so)
741.01 741.01 1.86 open64 (libpthread-2.11.3.so)
598.43 598.43 1.50 read (libpthread-2.11.3.so)
63.38 63.38 0.15 close (libpthread-2.11.3.so)
2.264 2.26 0.01 __lseek64 (libpthread-2.11.3.so)

openss>>expview -v calltrees,fullstack

Exclusive Inclusive % of Call Stack Function (defining location)
I/O call I/O call Total
times in times in Exclusive
seconds. seconds. CPU Time
TestIoSys (IOR: IOR.c,1848)
> @ 2608 in WriteOrRead (IOR: IOR.c,2562)
>> @ 244 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
38297.33 38297.33 96.46 >>>__write (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 2611 in WriteOrRead (IOR: IOR.c,2562)
>> @ 251 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
598.43 598.43 1.51 >>>read (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 104 in IOR_Create_POSIX (IOR: aiori-POSIX.c,74)
472.14 472.14 1.19 >>open64 (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 195 in IOR_Open_POSIX (IOR: aiori-POSIX.c,173)
268.88 268.88 0.68 >>open64 (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 316 in IOR_Close_POSIX (IOR: aiori-POSIX.c,315)
61.587482 61.587482 0.155123 >>close (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 316 in IOR_Close_POSIX (IOR: aiori-POSIX.c,315)
1.796442 1.796442 0.004525 >>close (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 2608 in WriteOrRead (IOR: IOR.c,2562)
>> @ 234 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
1.280113 1.280113 0.003224 >>>__lseek64 (libpthread-2.11.3.so)
TestIoSys (IOR: IOR.c,1848)
> @ 2611 in WriteOrRead (IOR: IOR.c,2562)
>> @ 234 in IOR_Xfer_POSIX (IOR: aiori-POSIX.c,224)
0.981341 0.981341 0.002472 >>>__lseek64 (libpthread-2.11.3.so)

```

In the above command line interface output, the expview command with no options gives the overview or summary view for all the ranks and threads. One can view the performance information for individual ranks (using -r <rank number>) or

individual threads (using `-t <thread number>`) or individual processes (using `-p <process id>`). One can also give a range of ranks, threads, or processes using their respective option.

For the calltrees view, the display is showing where the I/O function was called from in the user's application source. In this example, most of I/O time was spent in the write I/O function along the path shown in the first individual call path. The call path with fullstack option forces the calltrees view to not collapse any similar subtrees, which makes the view more explicit. Without the fullstack option the calltrees would be more consolidated.

## 6 Applying Experiments to Parallel Codes

The ideal scenario for the execution of parallel code using pthreads or OpenMP is efficient threading, where all threads are assigned work that can execute concurrently. Or for MPI code, the job is properly load balanced so all MPI ranks do the same amount of work and no MPI rank is stuck waiting.

What are some things that can cause these ideal scenarios to fail? (taken from LLNL parallel processing tutorial) MPI jobs can become unbalanced if an equal amount of work was not assigned to each rank, possibly through the number of array operations not being equal for each rank or loop iterations not being evenly distributed. You can still have problems even if your work seems to be evenly distributed. For example, if you evenly distribute a sparsely populated array then some ranks may end up with very little or no work while others will have a full workload. With adaptive grid models some ranks need to redefine their mesh while others don't. With N-body simulations some work migrates to other ranks so those ranks will have more to do while the others have less.

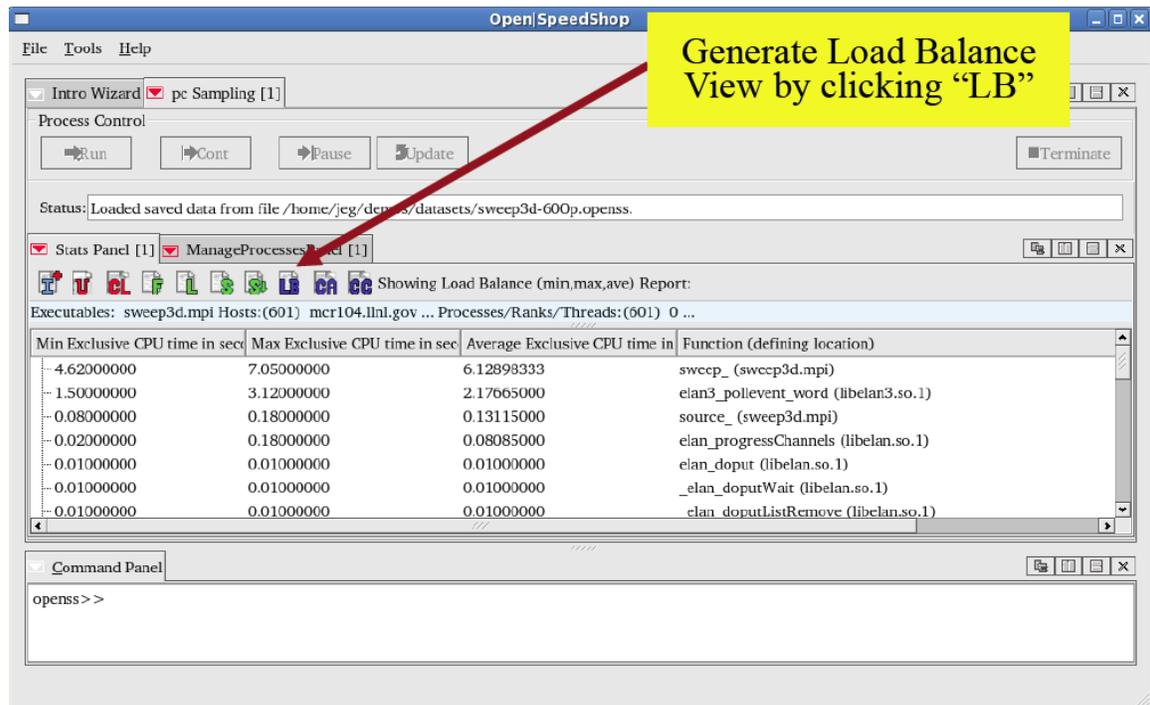
Performance analysis can help you with load balancing and an even distribution of work. Tools like O|SS are designed to work on parallel jobs. It supports threading and message passing and automatically tracks all ranks and threads during execution. It can also store the performance info per process, rank or thread for individual evaluation. All of the experiments for O|SS can be run on parallel jobs, collectors are applied to all ranks on all nodes. The results of an experiment can be displayed as an aggregation across all ranks or threads, which is the default view, or you can select individual or groups of ranks or threads to view. There are also experiments specifically designed for tracing MPI function calls.

O|SS has been tested with a variety of MPI versions including Open MPI, MVAPICH[2] and MPICH2 on Intel, Blue Gene, and Cray systems. O|SS is able to identify the MPI task (rank info) through the MPIR interface for the online version or through PMPI preload for the offline version. To run MPI code with O|SS just include the MPI launcher as part of the executable as normal, below are several examples:

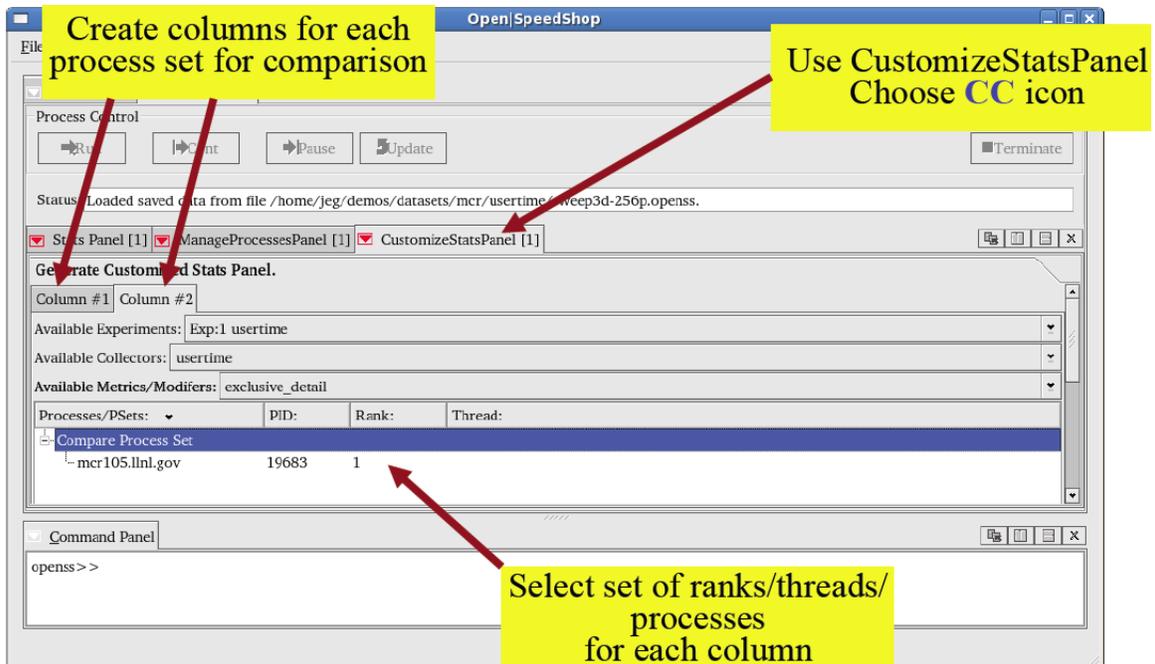
```
> ossmpi "mpirun -np 128 sweep3d.mpi"
> osspcsamp "mpirun -np 32 sweep3d.mpi"
> ossio "srun -N 4 -n 16 sweep3d.mpi"
> openss -offline -f "mpirun -np 128 sweep3d.mpi" hwctime
> openss -online -f "srun -N 8 -n 128 sweep3d.mpi" usertime
```

The default view for parallel applications is to aggregate the information collected across all ranks. You can manually include or exclude individual ranks, processes or

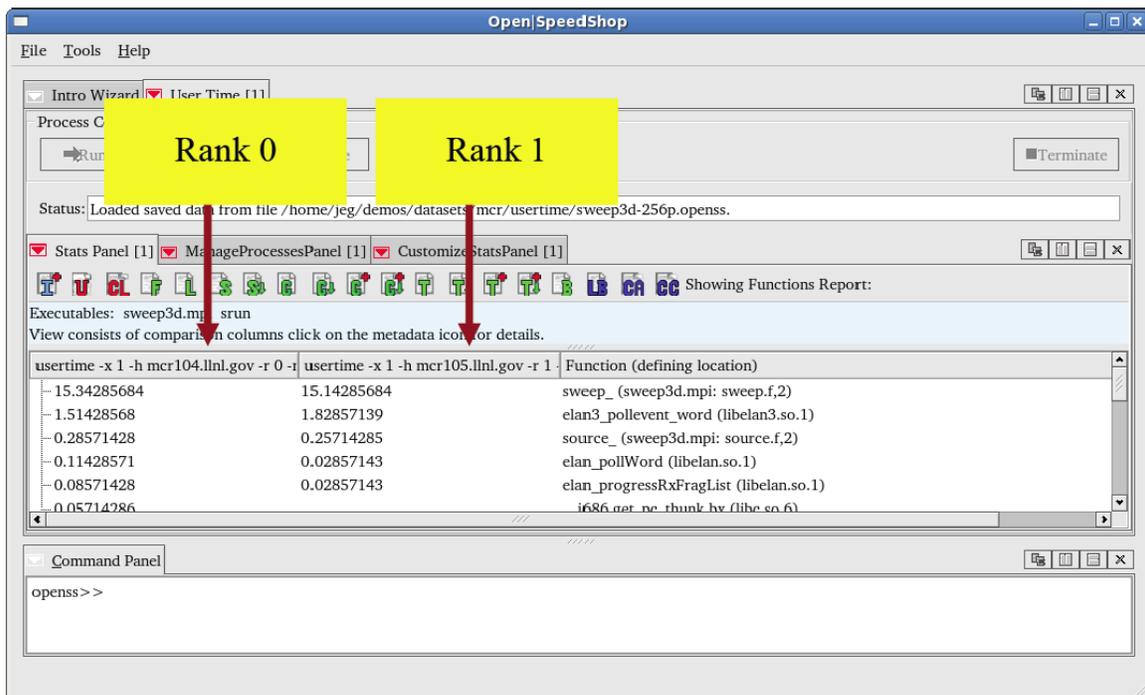
threads to view their specific results. You can also compare ranks by using the Customize Stats Panel View and creating a compare column for the process groups or individual ranks. Cluster analysis is also available, it can be used to find outliers, ranks that are performing very differently than the others. From the Stats Panel toolbar or context menu you can automatically create groups of similar performing ranks or threads. Through the Stat Panel O|SS also provides common analysis functions designed for quick analysis of MPI applications. There are load balance views that calculate min, max and average values across ranks, processes or threads. The image below shows the O|SS buttons for Load Balance and next to that Cluster Analysis.



Below we see the creation of a comparison between to ranks in O|SS.



Now we see those two ranks compared side by side in the statistics panel.



## 7 MPI Tracing Experiments (mpi, mpit, mpij)

In this section we will go through an MPI tracing experiment with O|SS. The experiment will record all MPI call invocations. There are three MPI experiments and associated convenience scripts, ossmpi, which will record call times ossmpit,

which will record call times and arguments, and mpip which is a lightweight version of of mpi, where individual MPI calls are recorded but not saved in the database. Equal events will be aggregated to save space in the database as well as to reduce the overhead.

Again we will run experiment on the smg2000 application. The syntax for the experiment is:

```
> ossmpi[t][p] "srun -N 4 -n 32 smg2000 -n 50 50 50" [default | <list MPI functions> | mpi_category]
```

The default behavior is to trace all MPI functions, but a comma separated list of MPI functions can be giving if you only want to trace specific functions, e.g. MPI\_Send, MPI\_Recv..., etc. You can also select an mpi\_category to trace: "all", "asynchronous\_p2p", "collective\_com", "datatypes", "environment", "graphs\_contexts\_comms", "persistent\_com", "process\_topologies", and "synchronous\_p2p".

The default views are designed to relate the information, included in the report, back to the individual calls to their corresponding MPI functions. This is the same information that would be reported if the user were to do an: "expview -m min, max, average". The view is a representation of the minimum, maximum and average time values per individual calls to their corresponding MPI functions.

The average time reported is the total amount of time for all the calls to a function divided by the total number of calls. Thus, it is the average time that each individual call spends in the function. As such, it is comparable to the Max (maximum) and Min (minimum) of a call to the function that is in the same "min, max, average" report.

Alternatively, if a user does an "expview -m ThreadMin, ThreadMax, ThreadAve", then the report information is related for the Max, Min and Average is related back to the individual ranks.

Another way of saying it is: The average is the total amount of time for all the calls to a function divided by the total number of ranks. Thus it is the average time that each rank spends in the function. As such, it is comparable to the Max and Min of a rank that is in the same report.

If the number of ranks is the same as the number of calls, the two different calculations should produce the same result. This would be true if all the calls were in a single thread or there were one in each rank, as it is for MPI\_Init.

The "expview -m min, max, average" view can expose load imbalance by showing when the minimum and maximum time for asynchronous MPI functions have large differences. This situation indicates that some of the MPI asynchronous functions ran quickly (low minimum times) but some had to wait a long time to get started

(large maximum times). Many times the function calls that ran quickly were the last to arrive and actually are from ranks that are not running as well as the others, causing load imbalance and delays to the overall job execution speed. These ranks show better performance numbers in terms of the MPI function time, but that is only because they were the last to arrive at the internal barrier point and did not have to wait as long as the other MPI functions that arrived sooner, but had to wait for the other ranks to finally arrive.

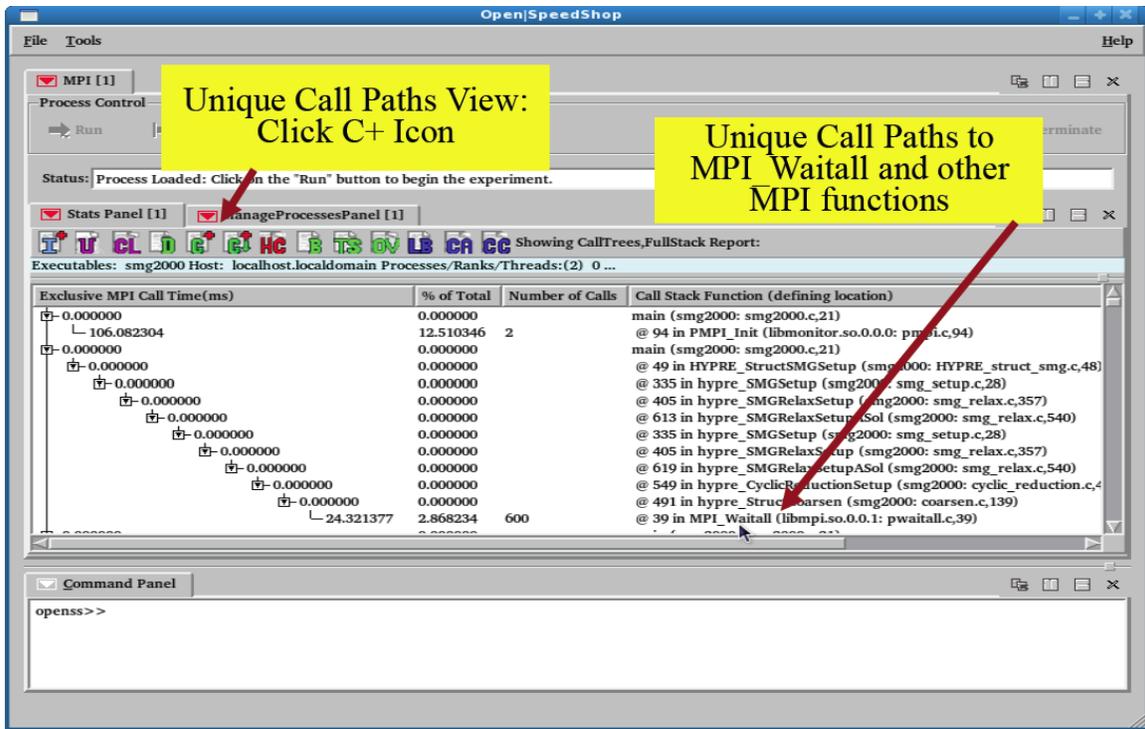
The image below shows the results of the MPI experiment in the default view.

Show min, max, average times per call.  
These numbers represent per function call time values.

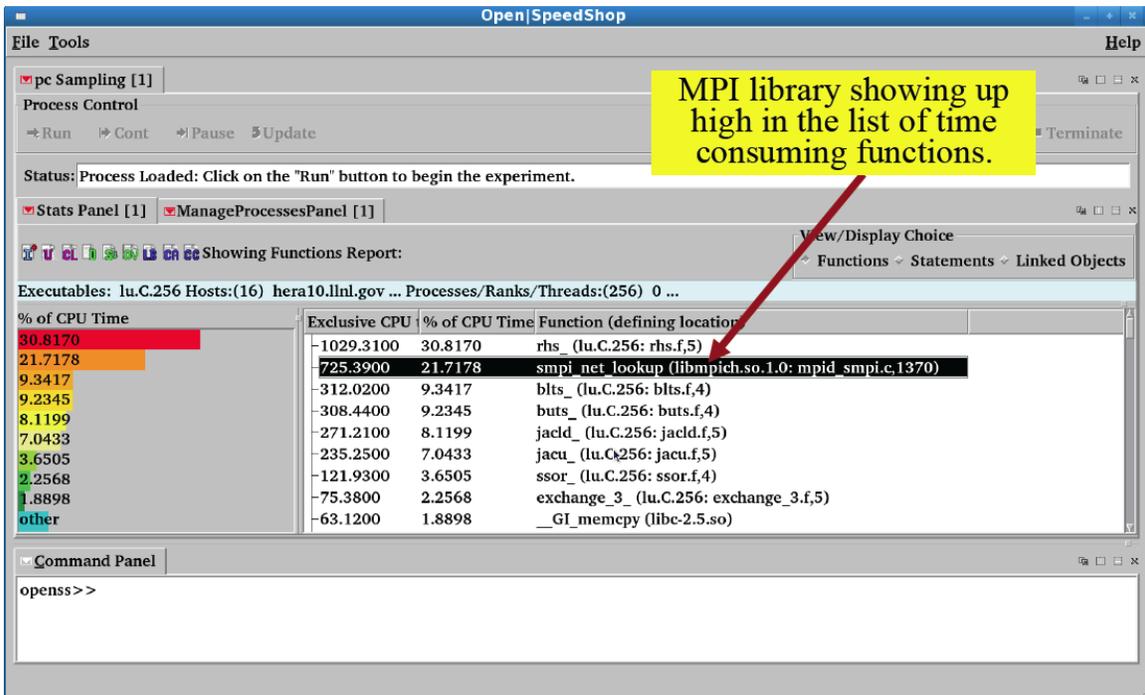
The screenshot shows the OpenSpeedShop interface with the MPI [1] process control panel. The 'Stats Panel [1]' is active, displaying a 'Showing Functions Report:'. Below the report, a table lists MPI function call times. A yellow callout box with an arrow points to the 'Minimum MPI Call Time(ms)', 'Maximum MPI Call Time(ms)', and 'Average Time(ms)' columns.

Exclusive MPI Call Time(ms)	% of MPI Total	Number of Calls	Minimum MPI Call Time(ms)	Maximum MPI Call Time(ms)	Average Time(ms)	Function (defining location)
-149558.7267	83.7791	1200	0.0169	2394.3711	124.6323	PMPI_Allreduce (libmpich.so.3.3)
-14002.3793	7.8438	798	0.0002	79.2830	17.5468	PMPI_Wait (libmpich.so.3.3)
-9767.5648	5.4715	6500	0.0005	25.2389	1.5027	PMPI_Bcast (libmpich.so.3.3)
-2849.7329	1.5964	13584	0.0002	17.6646	0.2098	MPI_Waitany (libmpich.so.3.3)
-1190.0472	0.6666	776	0.0002	36.5529	1.5336	MPI_Waitall (libmpich.so.3.3)
-982.9403	0.5506	392	0.0003	33.6511	2.5075	PMPI_Recv (libmpich.so.3.3)
-88.0546	0.0493	100	0.0021	18.1551	0.8805	MPI_Reduce (libmpich.so.3.3)
-32.2866	0.0181	13986	0.0004	0.6326	0.0023	PMPI_Isend (libmpich.so.3.3)
-28.9642	0.0162	398	0.0010	0.5800	0.0728	PMPI_Rsend (libmpich.so.3.3)
-12.3708	0.0069	14784	0.0001	0.9621	0.0008	MPI_Irecv (libmpich.so.3.3)

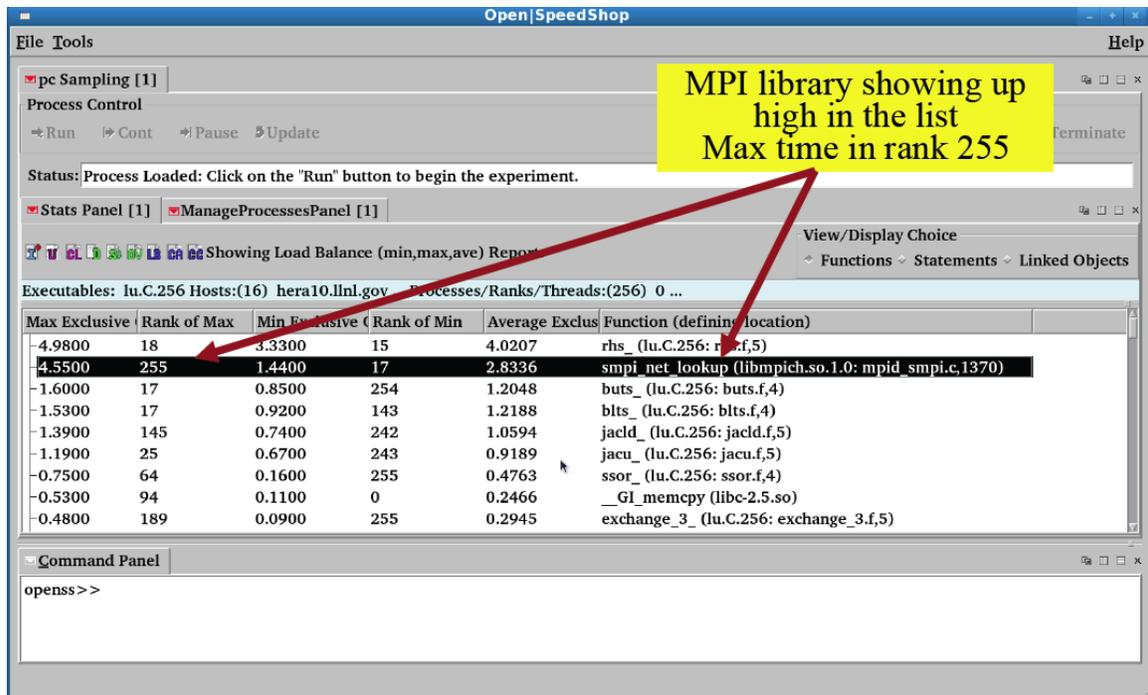
Next we see the MPI function call path view, shown below.



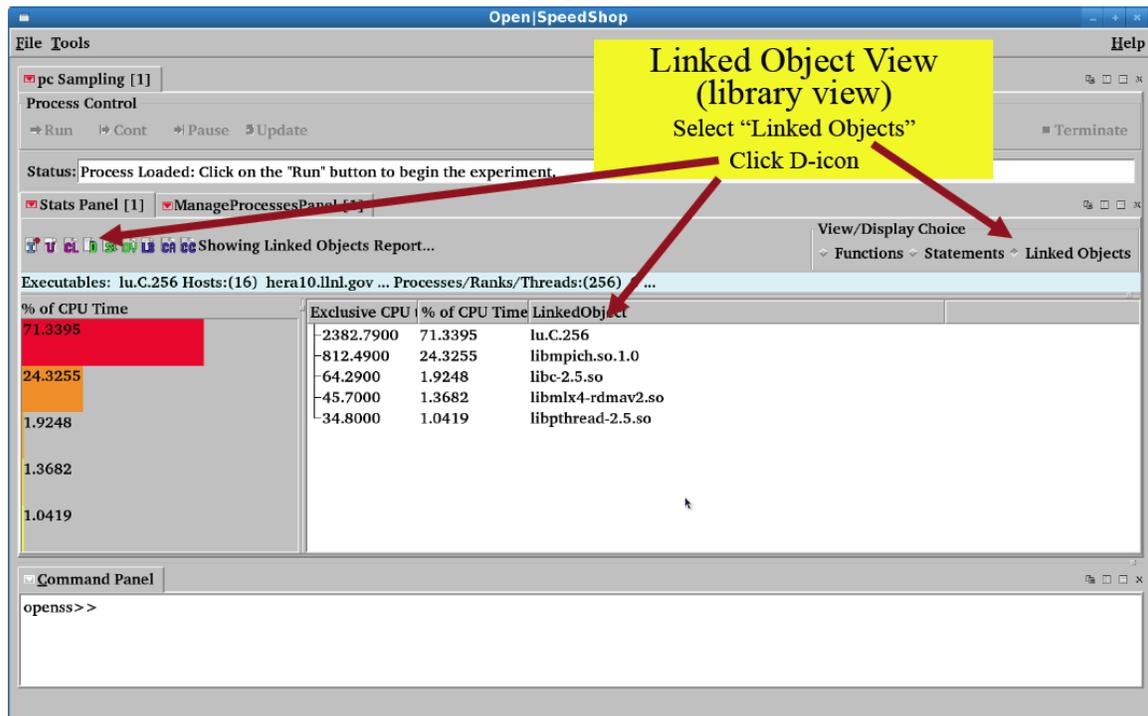
Here is the default pcsamp view based on functions.



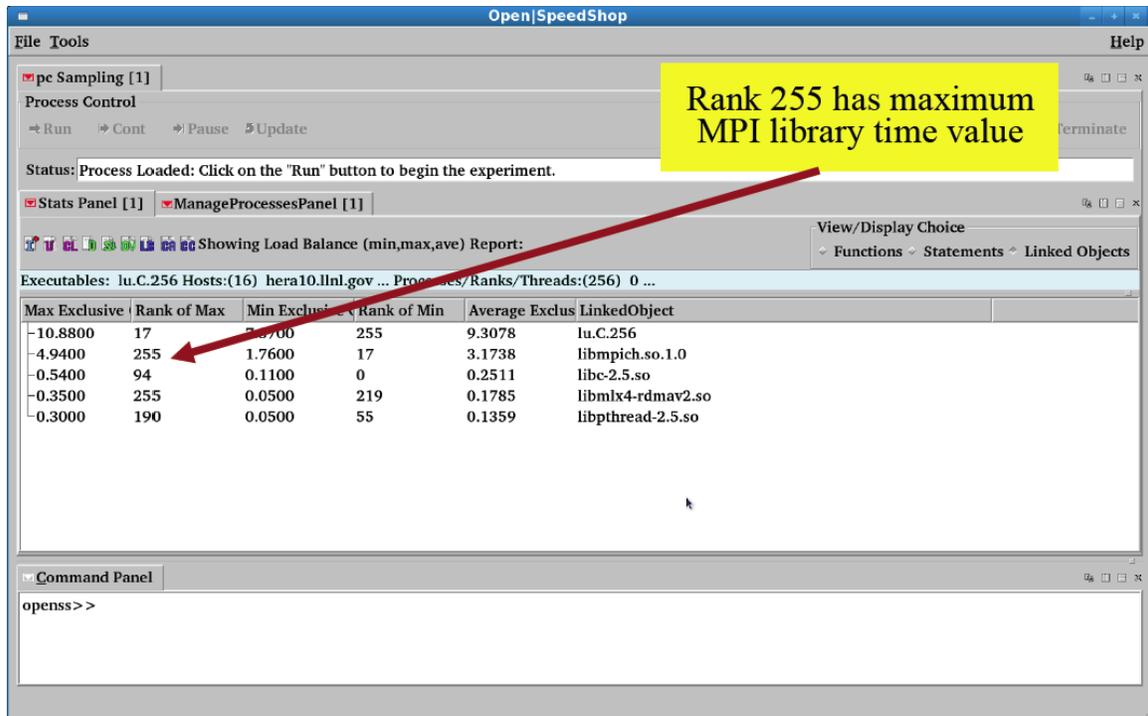
Here is the load balance view based on functions.



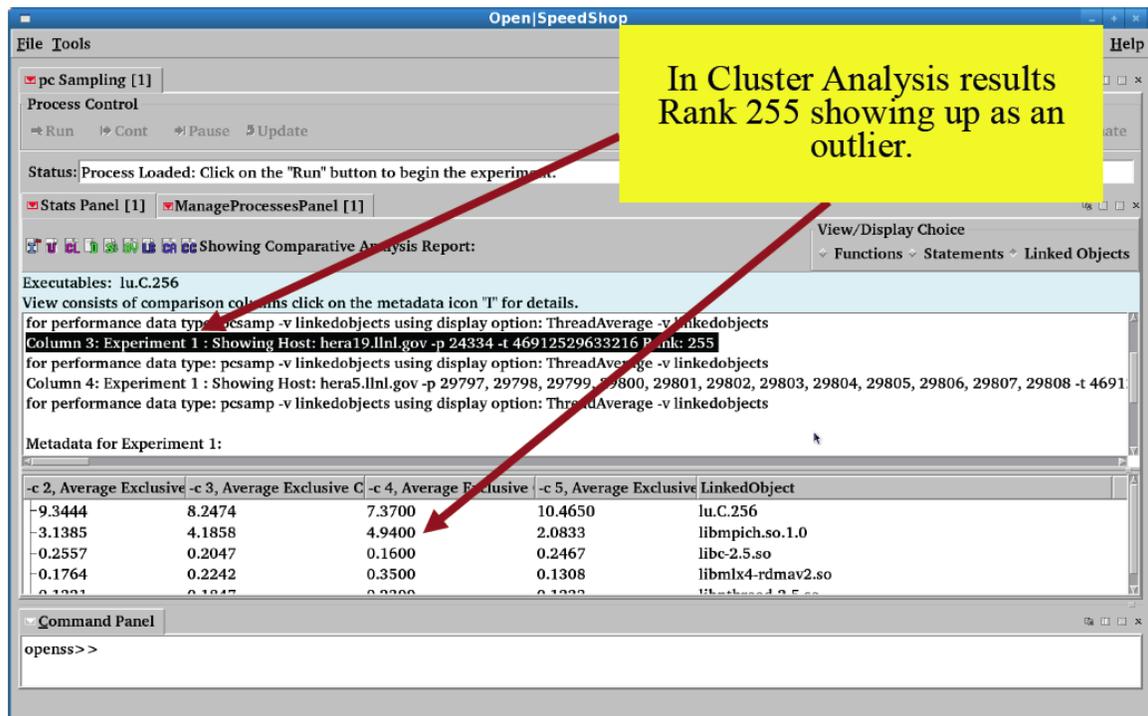
Here is the default view based on Linked Objects (libraries).



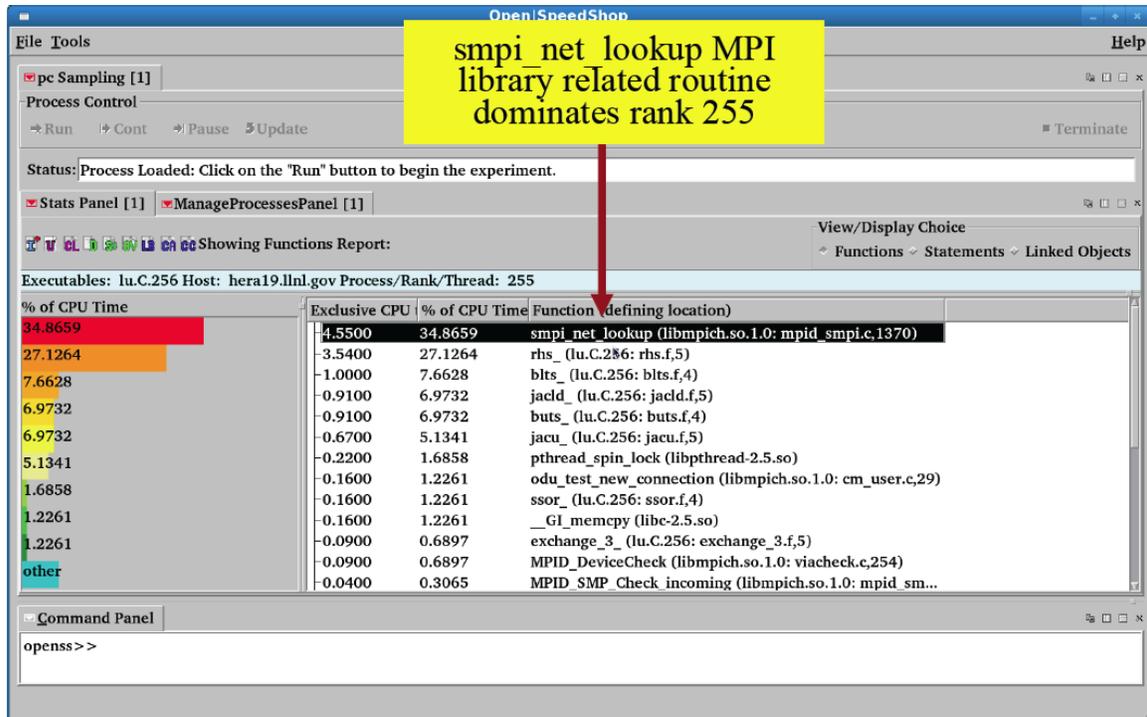
Next we see the load balance view base on Linked Objects (libraries).



Here we see the cluster analysis view based on Linked Objects.



Here is the pcsamp view of Rank 255 performance data only.



Below we examine Rank 255 further but this time using the load balance view in the Command Line Interface for OJSS.

openss>>expview -m loadbalance

Max MPI Call Time (defining location)	Rank of Max	Min MPI Call Time	Rank of Min	Average MPI Call Function
150332.97	0	120351.97	36	131361.13 MPI_Recv (libmpich.so.1.0: recv.c,60)
17636.11	36	1103.53	0	5443.08 MPI_Send (libmpich.so.1.0: send.c,65)
16470.53	19	353.81	0	5255.33 MPI_Wait (libmpich.so.1.0: wait.c,51)
3206.45	255	3.00	17	2000.27 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
915.17	54	754.39	83	792.07 PMP_Init (libmonitor.so.0.0.0: pmpi.c,94)
16.00	48	5.63	249	7.29 PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223)
9.28	230	2.90	0	7.55 MPI_Irecv (libmpich.so.1.0: irecv.c,48)
1.22	247	0.07	0	1.10 MPI_Bcast (libmpich.so.1.0: bcast.c,81)
0.51	53	0.35	239	0.41 MPI_Barrier (libmpich.so.1.0: barrier.c,56)

openss>>

MPI Experiment shows Rank 255 spending significant time in MPI\_Allreduce

Here we look at the difference between Rank 255 and Rank 0.

```
openss>>expview -r 255 -m exclusive_time
```

```
openss>>expview -r 0 -m exclusive_time
```

Exclusive MPI Call Function (defining location)

Time(ms)	Function (defining location)
138790.370000	MPI_Recv (libmpich.so.1.0: recv.c,60)
8841.088000	MPI_Wait (libmpich.so.1.0: wait.c,51)
3337.737000	MPI_Send (libmpich.so.1.0: send.c,65)
3206.454000	MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
797.964000	PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94)
5.887000	PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223)
4.701000	MPI_Irecv (libmpich.so.1.0: irecv.c,48)
1.221000	MPI_Bcast (libmpich.so.1.0: bcast.c,81)
0.396000	MPI_Barrier (libmpich.so.1.0: barrier.c,55)

Exclusive MPI Call Function (defining location)

Time(ms)	Function (defining location)
150332.974000	MPI_Recv (libmpich.so.1.0: recv.c,60)
1103.539000	MPI_Send (libmpich.so.1.0: send.c,65)
807.433000	PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94)
353.810000	MPI_Wait (libmpich.so.1.0: wait.c,51)
15.643000	PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223)
8.903000	MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
2.995000	MPI_Irecv (libmpich.so.1.0: irecv.c,48)
0.438000	MPI_Barrier (libmpich.so.1.0: barrier.c,56)
0.076000	MPI_Bcast (libmpich.so.1.0: bcast.c,81)

MPI Experiment comparison of rank 255 to another (rank 0) shows Rank 255 spending much more time in MPI\_Allreduce and MPI\_Wait

Next we see the hot call paths for MPI\_Wait on Rank 255.

```
openss>>expview -r 255 -vcalltrees,fullstack -f MPI_Wait
```

Exclusive MPI Call Time(ms)	% of Total	Number of Calls	Call Stack Function (defining location)
6010.978000	3.878405	1	>>>>main (lu.C.256) >>>> @ 140 in MAIN__ (lu.C.256: lu.f,46) >>>>> @ 180 in ssor_ (lu.C.256: ssor.f,4) >>>>>> @ 213 in rhs_ (lu.C.256: rhs.f,5) >>>>>>> @ 224 in exchange_3_ (lu.C.256: exchange_3.f,5) >>>>>>>> @ 893 in mpi_wait_ (mpi-mvapich-rt-offline.so: wrappers-fortran.c,893) >>>>>>>>> @ 889 in mpi_wait (mpi-mvapich-rt-offline.so: wrappers-fortran.c,885) 250 >>>>>>>>>> @ 51 in MPI_Wait (libmpich.so.1.0: wait.c,51)
2798.770000	1.805823	1	>>>>main (lu.C.256) >>>> @ 140 in MAIN__ (lu.C.256: lu.f,46) >>>>> @ 180 in ssor_ (lu.C.256: ssor.f,4) >>>>>> @ 64 in rhs_ (lu.C.256: rhs.f,5) >>>>>>> @ 88 in exchange_3_ (lu.C.256: exchange_3.f,5) >>>>>>>> @ 893 in mpi_wait_ (mpi-mvapich-rt-offline.so: wrappers-fortran.c,893) >>>>>>>>> @ 889 in mpi_wait (mpi-mvapich-rt-offline.so: wrappers-fortran.c,885) 250 >>>>>>>>>> @ 51 in MPI_Wait (libmpich.so.1.0: wait.c,51)

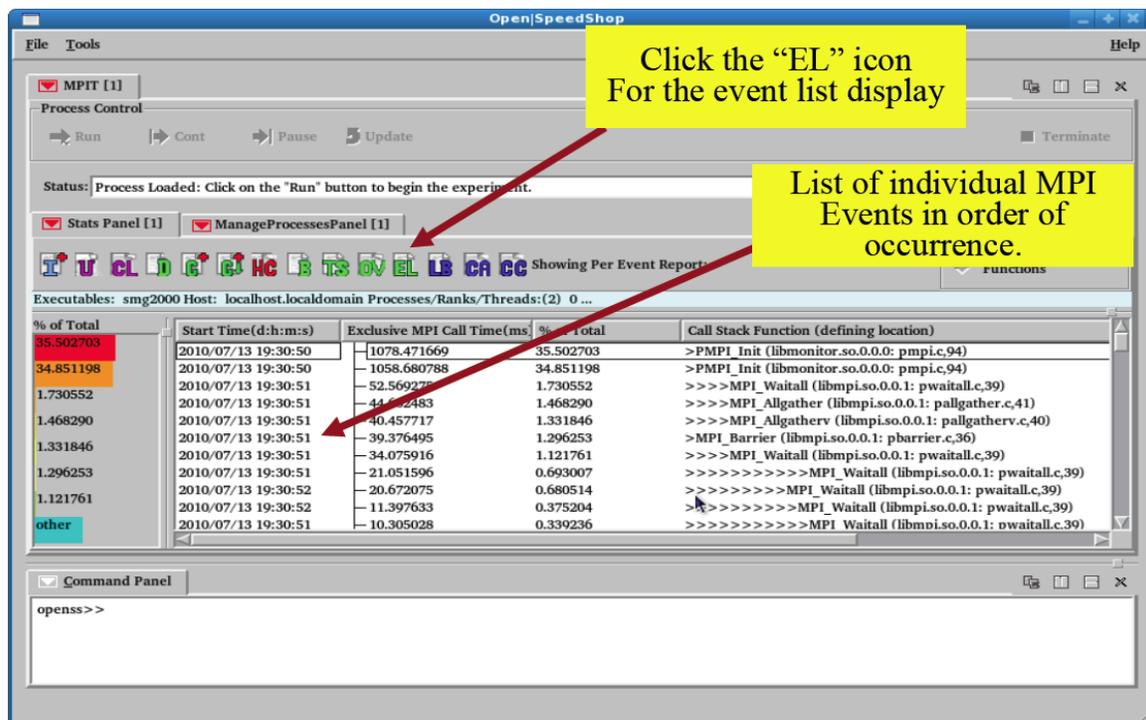
Most expensive call path to MPI\_Wait

In this experiment we did program counter sampling to get an overview of the application. We noticed that smp\_net\_lookup showed up in function load balance view, which caused us to take a look at the linked object view. The load balance on the linked object showed some imbalance, so we looked at the cluster analysis view and found that rank 255 was an outlier.

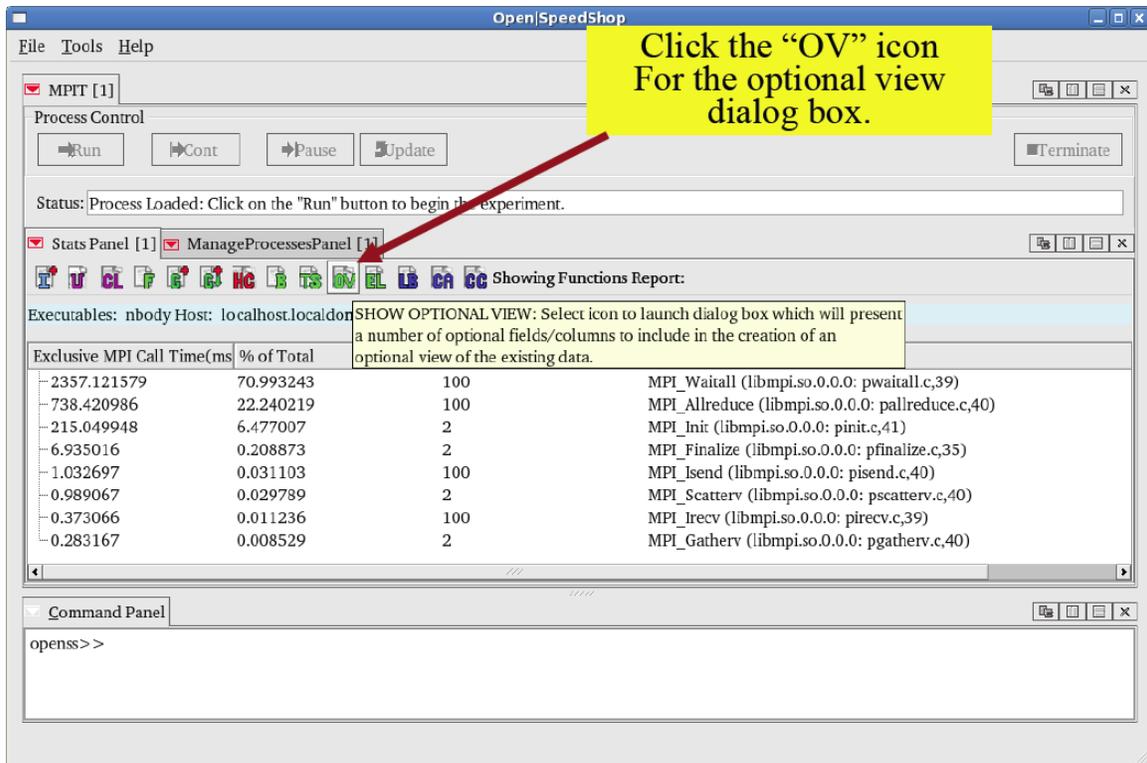
We then took a closer look at rank 255 and saw that the pcsamp output shows most of the time was spent in smp\_net\_lookup. We used the MPI experiment to determine if we can get more clues and saw that a load balance view on the MPI experiment shows rank 255's MPI\_Allreduce time is the highest of the 256 ranks. We then looked at rank 255 and a representative rank from the rest of the ranks and noted the differences in MPI\_Wait, MPI\_Send and MPI\_Allreduce. We looked at the call paths to MPI\_Wait to determine why the wait was occurring.

The mpit experiment has a performance information entry for each MPI function call. In addition to the time spent in each MPI function, information like source and destination rank, bytes sent or received are also available. You can selectively view the information you desire.

Below we see the default event view for an MPI application.



We can create our own event view with the OV button.

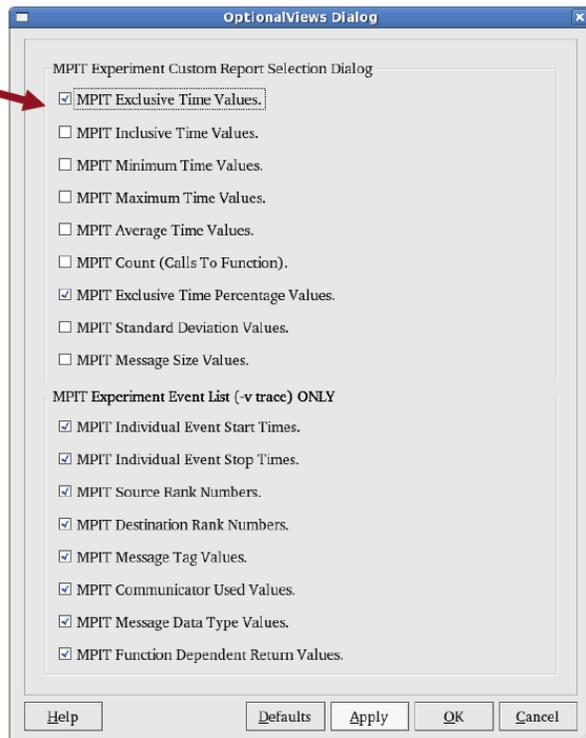


You can use the views dialog box to choose what metric to display.

Select the metric values  
you want to see in the  
display and click OK

Use the Optional Views Dialog  
box to choose the performance metrics  
to be displayed in the StatsPanel  
and click OK

Clicking OK will regenerate the  
StatsPanel with the new metrics  
displayed



After choosing the event to view it will then be displayed.



### 7.1.2 MPI Tracing Experiment (mpi) performance data viewing with GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

### 7.1.3 MPI Tracing Experiment (mpi) performance data viewing with CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”.

The following table describes the header and column data definitions for the default MPI experiment views.

Column Name	Column Definition
Exclusive MPI Call Time	Aggregated total exclusive time spent in the MPI function corresponding to this row of data.
% of MPI Time	Percentage of exclusive MPI time spent in the MPI function corresponding to this row of data relative to the total MPI time for all the MPI functions.
Number of Calls	Total number of calls to the MPI function corresponding to this row of data.
Min MPI Call Time	The minimum time that a MPI call took across all calls spent in the corresponding MPI function.
Max MPI Call Time	The maximum time that a MPI call took across all calls spent in the corresponding MPI function.
Average MPI Call Time Across Ranks	The average time for the default view is the total amount of time for all the calls to a function divided by the total number of calls. Thus, it is the average time that each MPI function call spends in the function.

This is an example of the CLI default view for the MPI (mpi, mpit) experiments.

```

jcg@localhost:~/ucar
File Edit View Search Terminal Help
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview

Exclusive   % of   Number   Minimum   Maximum   Average   Function (defining location)
MPI Call    MPI    of        MPI        MPI Call   Time(ms)
Time(ms)   Total  Calls     Call      Time(ms)
Time(ms)

149558.7267 83.7791 1200      0.0169    2394.3711 124.6323  PMPI_Allreduce (libmpich.so.3.3)
14002.3793  7.8438  798      0.0002    79.2830   17.5468  PMPI_Wait (libmpich.so.3.3)
9767.5648   5.4715  6500     0.0005    25.2389   1.5027   PMPI_Bcast (libmpich.so.3.3)
2849.7329   1.5964  13584    0.0002    17.6646   0.2098   MPI_Waitany (libmpich.so.3.3)
1190.0472   0.6666  776      0.0002    36.5529   1.5336   MPI_Waitall (libmpich.so.3.3)
982.9403    0.5506  392      0.0003    33.6511   2.5075   PMPI_Recv (libmpich.so.3.3)
88.0546     0.0493  100      0.0021    18.1551   0.8805   MPI_Reduce (libmpich.so.3.3)
32.2866     0.0181  13986    0.0004    0.6326    0.0023   PMPI_Isend (libmpich.so.3.3)
28.9642     0.0162  398      0.0010    0.5800    0.0728   PMPI_Rsend (libmpich.so.3.3)
12.3708     0.0069  14784    0.0001    0.9621    0.0008   MPI_Irecv (libmpich.so.3.3)
2.4564      0.0014  792      0.0006    0.0169    0.0031   PMPI_Send (libmpich.so.3.3)
openss>>

```

This is an example of the GUI default view for the MPI (mpi, mpit) experiments.

Executables: csm.exe Hosts:(14) ys6014 ... Pids: 200 Ranks: 200 Threads: 200

Exclusive MPI Call Time(ms)	% of MPI Total	Number of Calls	Minimum MPI Call Time(ms)	Maximum MPI Call Time(ms)	Average Time(ms)	Function (defining location)
-14958.7267	83.7791	1200	0.0169	2394.3711	124.6323	PMPI_Allreduce (libmpich.so.3.3)
-14002.3793	7.8438	798	0.0002	79.2830	17.5468	PMPI_Wait (libmpich.so.3.3)
-9767.5648	5.4715	6500	0.0005	25.2389	1.5027	PMPI_Bcast (libmpich.so.3.3)
-2849.7329	1.5964	13584	0.0002	17.6646	0.2098	MPI_Waitany (libmpich.so.3.3)
-1190.0472	0.6666	776	0.0002	36.5529	1.5336	MPI_Waitall (libmpich.so.3.3)
-982.9403	0.5506	392	0.0003	33.6511	2.5075	PMPI_Recv (libmpich.so.3.3)
-88.0546	0.0493	100	0.0021	18.1551	0.8805	MPI_Reduce (libmpich.so.3.3)
-32.2866	0.0181	13986	0.0004	0.6326	0.0023	PMPI_Isend (libmpich.so.3.3)
-28.9642	0.0162	398	0.0010	0.5800	0.0728	PMPI_Rsend (libmpich.so.3.3)
-12.3708	0.0069	14784	0.0001	0.9621	0.0008	MPI_Recv (libmpich.so.3.3)

The default views are designed to relate the information, included in the report, back to the individual calls to their corresponding MPI functions. This is the same information that would be reported if the user were to do: “expview -m min, max, average”. The view is a representation of the minimum, maximum and average time values per individual calls to their corresponding MPI functions.

The average time reported is the total amount of time for all the calls to a function divided by the total number of calls. Thus, it is the average time that each individual call spends in the function. As such, it is comparable to the Max (maximum) and Min (minimum) of a call to the function that is in the same “min, max, average” report.

Alternatively, if a user does an “expview -m ThreadMin, ThreadMax, ThreadAve”, then the report information is related for the Max, Min and Average is related back to the individual ranks.

Another way of saying it is: The average is the total amount of time for all the calls to a function divided by the total number of ranks. Thus it is the average time that each rank spends in the function. As such, it is comparable to the Max and Min of a rank that is in the same report.

If the number of ranks is the same as the number of calls, the two different calculations should produce the same result. This would be true if all the calls were in a single thread or there were one in each rank, as it is for MPI\_Init.

The “expview -m min, max, average” view can expose load imbalance by showing when the minimum and maximum time for asynchronous MPI functions have large differences. This situation indicates that some of the MPI asynchronous functions ran quickly (low minimum times) but some had to wait a long time to get started (large maximum times). Many times the function calls that ran quickly were the last to arrive and actually are from ranks that are not running as well as the others,

causing load imbalance and delays to the overall job execution speed. These ranks show better performance numbers in terms of the MPI function time, but that is only because they were the last to arrive at the internal barrier point and did not have to wait as long as the other MPI functions that arrived sooner, but had to wait for the other ranks to finally arrive.

## 7.2 MPI Tracing Experiments (mpit)

### 7.2.1 MPI Tracing Experiments (mpit) performance data gathering

Much of this information is described above in the main MPI Tracing Experiments section, but for completeness, this is the convenience script description for running the MPI specific tracing experiments.

```
> ossmpit "srun -N 4 -n 32 smg2000 -n 50 50 50" [default | <list MPI functions> | mpi_category]
```

If the mpi\_category or a list of categories is given to the ossmpit command, then only those the MPI function corresponding to that category will be traced. The MPI categories are defined according to the table below.

MPI Category	Argument
All MPI Functions	all
Collective Communicators	collective_com
Persistent Communicators	persistent_com
Synchronous Point to Point	synchronous_p2p
Asynchronous Point to Point	asynchronous_p2p
Process Topologies <sup>[1]</sup> <sub>SEP</sub>	process_topologies
Groups Contexts Communicators	graphs_contexts_comms
Environment <sup>[1]</sup> <sub>SEP</sub>	environment <sup>[1]</sup> <sub>SEP</sub>
Datatypes	datatypes
MPI File I/O	fileio

### 7.2.2 MPI Tracing Experiments (mpit) performance data viewing with GUI

To launch the GUI on any experiment, use “openss -f <database name>”.

### 7.2.3 MPI Tracing Experiments (mpit) performance data viewing with CLI

To launch the CLI on any experiment, use “openss -cli -f <database name>”.

The following table describes the header and column data definitions for the default MPI experiment views.

Column Name	Column Definition
Exclusive MPI Call Time	Aggregated total exclusive time spent in the MPI function corresponding to this row of data.

Column Name	Column Definition
% of MPI Time	Percentage of exclusive MPI time spent in the MPI function corresponding to this row of data relative to the total MPI time for all the MPI functions.
Number of Calls	Total number of calls to the MPI function corresponding to this row of data.
Min MPI Call Time	The minimum time that a MPI call took across all calls spent in the corresponding MPI function.
Max MPI Call Time	The maximum time that a MPI call took across all calls spent in the corresponding MPI function.
Average MPI Call Time Across Ranks	The average time for the default view is the total amount of time for all the calls to a function divided by the total number of calls. Thus, it is the average time that each MPI function call spends in the function.

This is an example of the CLI default view for the MPI (mpi, mpit) experiments.

```

jcg@localhost:~/ucar
File Edit View Search Terminal Help
opense>>[opense]: The restored experiment identifier is: -x 1
opense>>expview

Exclusive MPI Call Time(ms)  % of MPI Total  Number of Calls  Minimum MPI Call Time(ms)  Maximum MPI Call Time(ms)  Average Time(ms)  Function (defining location)
-----
149558.7267  83.7791  1200  0.0169  2394.3711  124.6323  PMPI_Allreduce (libmpich.so.3.3)
14002.3793  7.8438  798  0.0002  79.2830  17.5468  PMPI_Wait (libmpich.so.3.3)
9767.5648  5.4715  6500  0.0005  25.2389  1.5027  PMPI_Bcast (libmpich.so.3.3)
2849.7329  1.5964  13584  0.0002  17.6646  0.2098  MPI_Waitany (libmpich.so.3.3)
1190.0472  0.6666  776  0.0002  36.5529  1.5336  MPI_Waitall (libmpich.so.3.3)
982.9403  0.5506  392  0.0003  33.6511  2.5075  PMPI_Recv (libmpich.so.3.3)
88.0546  0.0493  100  0.0021  18.1551  0.8805  MPI_Reduce (libmpich.so.3.3)
32.2866  0.0181  13986  0.0004  0.6326  0.0023  PMPI_Isend (libmpich.so.3.3)
28.9642  0.0162  398  0.0010  0.5800  0.0728  PMPI_Rsend (libmpich.so.3.3)
12.3708  0.0069  14784  0.0001  0.9621  0.0008  MPI_Irecv (libmpich.so.3.3)
2.4564  0.0014  792  0.0006  0.0169  0.0031  PMPI_Send (libmpich.so.3.3)
opense>>

```

This is an example of the GUI default view for the MPI (mpi, mpit) experiments.

The screenshot shows the OpenSpeedShop GUI interface. At the top, there's a menu bar with 'File', 'Tools', and 'Help'. Below that is a 'Process Control' section with buttons for 'Run', 'Ctrl', 'Pause', 'Update', and 'Terminate'. A status bar indicates 'Process Loaded: Click on the "Run" button to begin the experiment.' Below this is a 'Stats Panel' and a 'View/Display Choice' dropdown set to 'Functions'. The main area displays a table of MPI call statistics, which is identical to the CLI output shown in the previous block. At the bottom, there is a 'Command Panel' with the prompt 'opense>>'.

The default views are designed to relate the information, included in the report, back to the individual calls to their corresponding MPI functions. This is the same information that would be reported if the user were to do: “expview -m min, max, average”. The view is a representation of the minimum, maximum and average time values per individual calls to their corresponding MPI functions.

The average time reported is the total amount of time for all the calls to a function divided by the total number of calls. Thus, it is the average time that each individual call spends in the function. As such, it is comparable to the Max (maximum) and Min (minimum) of a call to the function that is in the same “min, max, average” report.

Alternatively, if a user does an “expview -m ThreadMin, ThreadMax, ThreadAve”, then the report information is related for the Max, Min and Average is related back to the individual ranks.

Another way of saying it is: The average is the total amount of time for all the calls to a function divided by the total number of ranks. Thus it is the average time that each rank spends in the function. As such, it is comparable to the Max and Min of a rank that is in the same report.

If the number of ranks is the same as the number of calls, the two different calculations should produce the same result. This would be true if all the calls were in a single thread or there were one in each rank, as it is for MPI\_Init.

The “expview -m min, max, average” view can expose load imbalance by showing when the minimum and maximum time for asynchronous MPI functions have large differences. This situation indicates that some of the MPI asynchronous functions ran quickly (low minimum times) but some had to wait a long time to get started (large maximum times). Many times the function calls that ran quickly were the last to arrive and actually are from ranks that are not running as well as the others, causing load imbalance and delays to the overall job execution speed. These ranks show better performance numbers in terms of the MPI function time, but that is only because they were the last to arrive at the internal barrier point and did not have to wait as long as the other MPI functions that arrived sooner, but had to wait for the other ranks to finally arrive.

## **7.3 MPI Tracing Experiments (mpip)**

### **7.3.1 MPI Tracing Experiments (mpip) performance data gathering**

Much of this information is described above in the main MPI Tracing Experiments section, but for completeness, this is the convenience script description for running the MPI specific (mpi, mpit, mpip) tracing experiments.

> ossmpip "srun -N 4 -n 32 smg2000 -n 50 50 50" [default | <list MPI functions> | mpi\_category]

If the mpi\_category or a list of categories is given to the ossmpi command, then only those the MPI function corresponding to that category will be traced. The MPI categories are defined according to the table below.

MPI Category	Argument
All MPI Functions	all
Collective Communicators	collective_com
Persistent Communicators	persistent_com
Synchronous Point to Point	synchronous_p2p
Asynchronous Point to Point	asynchronous_p2p
Process Topologies <sup>[1]</sup> <sub>[SEP]</sub>	process_topologies
Groups Contexts Communicators	graphs_contexts_comms
Environment <sup>[1]</sup> <sub>[SEP]</sub>	environment <sup>[1]</sup> <sub>[SEP]</sub>
Datatypes	datatypes
MPI File I/O	fileio

### 7.3.2 MPI Tracing Experiments (mpip) performance data viewing with CLI

To launch the CLI on any experiment, use "openss -cli -f <database name>".

The following table describes the header and column data definitions for the default MPI experiment views.

Column Name	Column Definition
Exclusive MPI Call Time	Aggregated total exclusive time spent in the MPI function corresponding to this row of data.
% of MPI Time	Percentage of exclusive MPI time spent in the MPI function corresponding to this row of data relative to the total MPI time for all the MPI functions.
Number of Calls	Total number of calls to the MPI function corresponding to this row of data.
Min MPI Call Time Across Ranks	The minimum time that a rank or ranks, across all ranks, spent in the corresponding MPI function.
Rank of Min	The number of the rank that had the minimum time spent in the MPI function across all the ranks of the application.

Max MPI Call Time Across Ranks	The maximum time that a rank or ranks, across all ranks, spent in the corresponding MPI function.
Rank of Max	The number of the rank that had the maximum time spent in the MPI function across all the ranks of the application.
Average MPI Call Time Across Ranks	The average for the default view is the total amount of time for all the calls to a function divided by the total number of ranks. Thus it is the average time that each rank spends in the function. As such, it is comparable to the Max and Min of a rank that is in the same report.

This is an example of the CLI default view for the MPI (mpip) experiments.

```

jgg@localhost:~/OpenSpeedShop_ROOT
File Edit View Search Terminal Help
opnsss>>
opnsss>>expview
Exclusive MPI call times in seconds, CPU Time
% of Total Exclusive Calls
Number of Calls
Min Rank of Exclusive Time Across Ranks(s)
Max Rank of Exclusive Time Across Ranks(s)
Average of Exclusive Time Across Ranks(s)
Function (defining location)
95034.514126 48.081511 27 3519.012910 23 3520.382349 6 3519.796819 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
48459.901000 24.517675 38907 380.920791 23 8518.191164 0 1794.811148 PMPI_Waitall (libmpich.so.1.0: waitall.c,57)
40505.338098 20.493161 12933 88.776030 16 3793.478559 0 1500.197522 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
12005.680319 6.194463 279676 33.093728 26 1061.876716 24 446.877345 PMPI_Wait (libmpich.so.1.0: wait.c,51)
1408.907554 0.712819 279676 11.513939 0 147.764633 14 52.181761 PMPI_Isend (libmpich.so.1.0: isend.c,58)
178.581476 0.090351 279676 4.602243 26 10.555328 13 6.614129 PMPI_Irecv (libmpich.so.1.0: irecv.c,48)
opnsss>>

```

This is an example of the CLI load balance view for the MPI (mpip) experiment. This view shows the minimum, maximum, and average time per rank for each function and the rank that represents the maximum time and minimum time. Note that there may be more ranks that have the same maximum and minimum time per rank.

```

jgg@localhost:~/OpenSpeedShop_ROOT
File Edit View Search Terminal Help
opnsss>>expview -m loadbalance
Exclusive MPI call times in seconds, CPU Time
% of Total Exclusive Calls
Number of Calls
Min Rank of Exclusive Time Across Ranks(s)
Max Rank of Exclusive Time Across Ranks(s)
Average of Exclusive Time Across Ranks(s)
Function (defining location)
8518.191164 0 380.920791 23 1794.811148 PMPI_Waitall (libmpich.so.1.0: waitall.c,57)
3793.478559 0 88.776030 16 1500.197522 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
3520.382349 6 3519.012910 23 3519.796819 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
1061.876716 24 33.093728 26 446.877345 PMPI_Wait (libmpich.so.1.0: wait.c,51)
147.764633 14 11.513939 0 52.181761 PMPI_Isend (libmpich.so.1.0: isend.c,58)
10.555328 13 4.602243 26 6.614129 PMPI_Irecv (libmpich.so.1.0: irecv.c,48)
opnsss>>

```

This is an example of the ability to compare performance information at the rank level in the CLI. Here we show a comparison on the exclusive time metric for rank 0 and rank 23. These ranks were show to be the ranks that had the maximum and minimum values for MPI\_Waitall above. One could also use the expview -r 0 and expview -r 23 to see the times for just those ranks.

```

jgg@localhost:~/OpenSpeedShop_ROOT
opnss>>expcmpare -r0,23 -mtime
-r 0, -r 23, Function (defining location)
Exclusive MPI call times in seconds
Exclusive MPI call times in seconds
8518.191164 380.920791 PMPI_Waitall (libmpich.so.1.0: waitall.c,57)
3793.478559 572.198026 MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
3519.665395 3519.012910 PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
699.083580 363.937877 PMPI_Wait (libmpich.so.1.0: wait.c,51)
11.513939 28.868632 PMPI_Isend (libmpich.so.1.0: isend.c,58)
5.608623 5.892431 PMPI_Irecv (libmpich.so.1.0: irecv.c,48)
opnss>>

```

Here we show the top two call paths in the program that took the most time (with respect to MPI function calls).

```

jgg@localhost:~/OpenSpeedShop_ROOT
opnss>>expview -v calltrees,fullstack mpip2
Exclusive MPI call times in seconds
% of Total CPU Time
Number of Calls
Min Exclusive Time Across Ranks(s)
Rank of Exclusive Time Across Ranks(s)
Average Exclusive Time Across Ranks(s)
Call Stack Function (defining location)
_start (lulesh)
> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 5340 in main (lulesh: lulesh.cc,5333)
>>>>> @ 104 in PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
_start (lulesh)
> @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> @ 258 in __libc_start_main (libc-2.12.so: libc-start.c,96)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 2381 in main (lulesh: lulesh.cc,5333)
>>>>> @ 59 in MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
95034.514126 48.081511 27
40505.333098 20.493161 12933
opnss>>

```

### 7.3.3 MPI Tracing Experiments (mpip) performance data viewing with GUI

To launch the GUI on any experiment, use “opnss -f <database name>”.

This is an example of the GUI default view for the MPI (mpip) experiment.

The screenshot shows the OpenSpeedShop GUI interface. At the top, it displays 'Open|SpeedShop (on rzmer1156)'. Below this, there are several panels:

- MPIP [1]**: Shows process control buttons (Run, Cont, Pause, Update) and a 'Terminate' button.
- Status**: Displays 'Process Loaded: Click on the "Run" button to begin the experiment.'
- Stats Panel [1]**: Shows 'Showing Functions Report:' and a 'View/Display Choice' dropdown set to 'Functions'.
- Executables**: Shows 'lulesh Hosts:(2) rzmer11 ... Pids: 27 Ranks: 27 Threads: 27'.
- Table**: A table with columns: Exclusive MPI call times in seconds, % of Total CPU Time, Number of Calls, Min Exclusive Time Across Ranks(s), Rank of Exclusive Time Across Ranks(s), Max Exclusive Time Across Ranks(s), Average Exclusive Time Across Ranks(s), and Function (defining location). The data rows are:
 

Exclusive MPI call times in seconds	% of Total CPU Time	Number of Calls	Min Exclusive Time Across Ranks(s)	Rank of Exclusive Time Across Ranks(s)	Max Exclusive Time Across Ranks(s)	Average Exclusive Time Across Ranks(s)	Function (defining location)
95034.514126	48.081511	27	3519.012910	23	3520.382349	6	3519.796819
48459.901000	24.517675	38907	380.920791	23	8518.191164	0	1794.811148
40505.333098	20.493161	12933	88.776030	16	3793.478559	0	1500.197522
12065.688319	6.104483	279676	33.093728	26	1061.876716	24	446.877345
1408.907554	0.712819	279676	11.513939	0	147.764633	14	52.181761
178.581476	0.090351	279676	4.602243	26	10.553328	13	6.614129
- Command Panel**: Shows 'opnss>>'.

The following view shows the load balance for this execution of lulesh on 27 ranks.

Showing Load Balance (min,max,ave) Report:

Max Exclusive Time Across	Rank of Max	Min Exclusive Time Across	Rank of Min	Average Exclusive Time Across	Function (defining location)
-8518.191164	0	380.920791	23	1794.811148	PMPI_Waitall (libmpich.so.1.0: waitall.c,57)
-3793.478559	0	88.776030	16	1500.197522	MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
-3520.382349	6	3519.012910	23	3519.796819	PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
-1061.876716	24	33.093728	26	446.877345	PMPI_Wait (libmpich.so.1.0: wait.c,51)
-147.764633	14	11.513939	0	52.181761	PMPI_Isend (libmpich.so.1.0: isend.c,58)
-10.555328	13	4.602243	26	6.614129	PMPI_Irecv (libmpich.so.1.0: irecv.c,48)

This view shows the cluster analysis view for this run of lulesh on 27 ranks. The cluster analysis view groups like performing ranks together as a means of locating groups of ranks that are outliers with respect to the other ranks.

Showing Comparative Analysis Report:

View consists of comparison columns click on the metadata icon "I" for details.

Comparing:

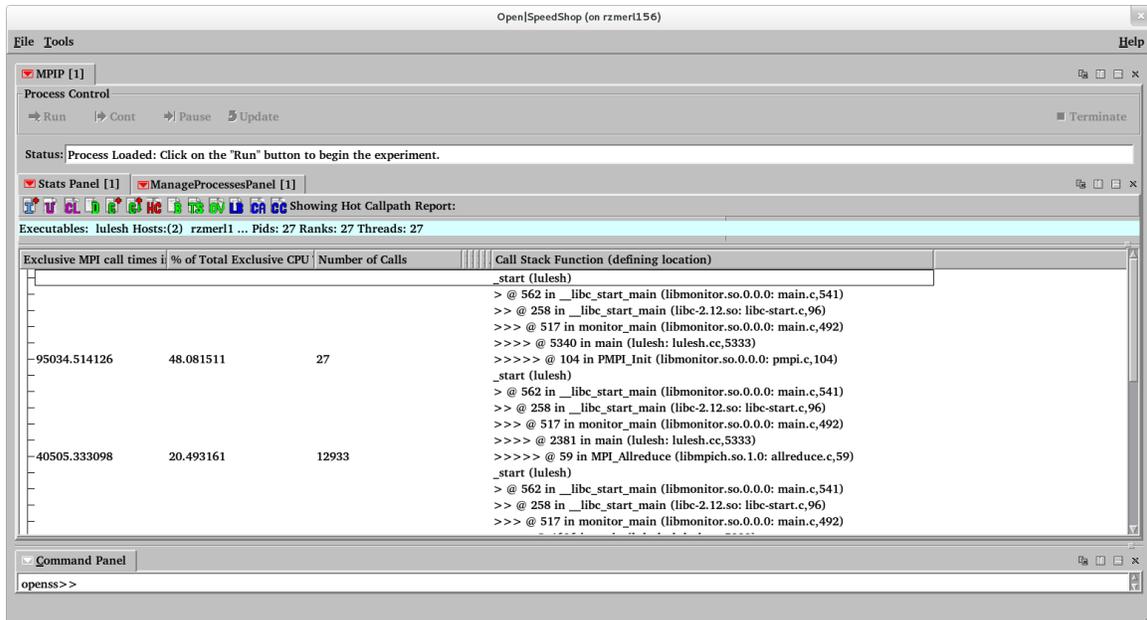
Column(s) labeled -c 2: Experiment 1 Database Name: lulesh-mpip-2.openss : Showing Host: rzmer11 0:46912593682880 Host: rzmer12 14:46912593682880 for performance data type: mpip -v functions using display option: ThreadAverage -v functions

Column(s) labeled -c 3: Experiment 1 Database Name: lulesh-mpip-2.openss : Showing Host: rzmer11 1:46912593682880 2:46912593682880 3:46912593682880 4:46912593682880 5:46912593682880 for performance data type: mpip -v functions using display option: ThreadAverage -v functions

Column(s) labeled -c 4: Experiment 1 Database Name: lulesh-mpip-2.openss : Showing Host: rzmer12 15:46912593682880 19:46912593682880 21:46912593682880 22:46912593682880 24:46912593682880 for performance data type: mpip -v functions using display option: ThreadAverage -v functions

-c 2, Average Exclusive Ti	-c 3, Average Exclusive Ti	-c 4, Average Exclusive Ti	Function (defining location)
-8050.206247	887.894809	2920.318465	PMPI_Waitall (libmpich.so.1.0: waitall.c,57)
-3519.417472	3519.823958	3519.840005	PMPI_Init (libmonitor.so.0.0.0: pmpi.c,104)
-3104.208653	914.826306	3200.077935	MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)
-567.671661	385.717683	643.198267	PMPI_Wait (libmpich.so.1.0: wait.c,51)
-79.639286	47.761277	58.880688	PMPI_Isend (libmpich.so.1.0: isend.c,58)
-6.855619	6.687907	6.222418	PMPI_Irecv (libmpich.so.1.0: irecv.c,48)

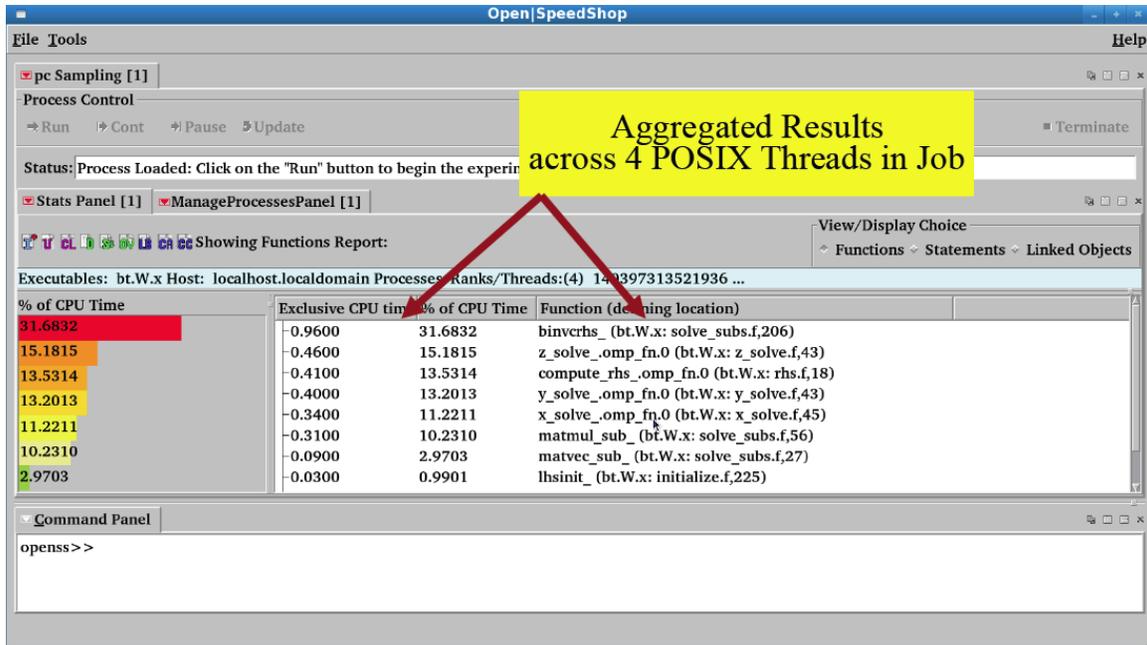
This view shows the hot call paths in the application.



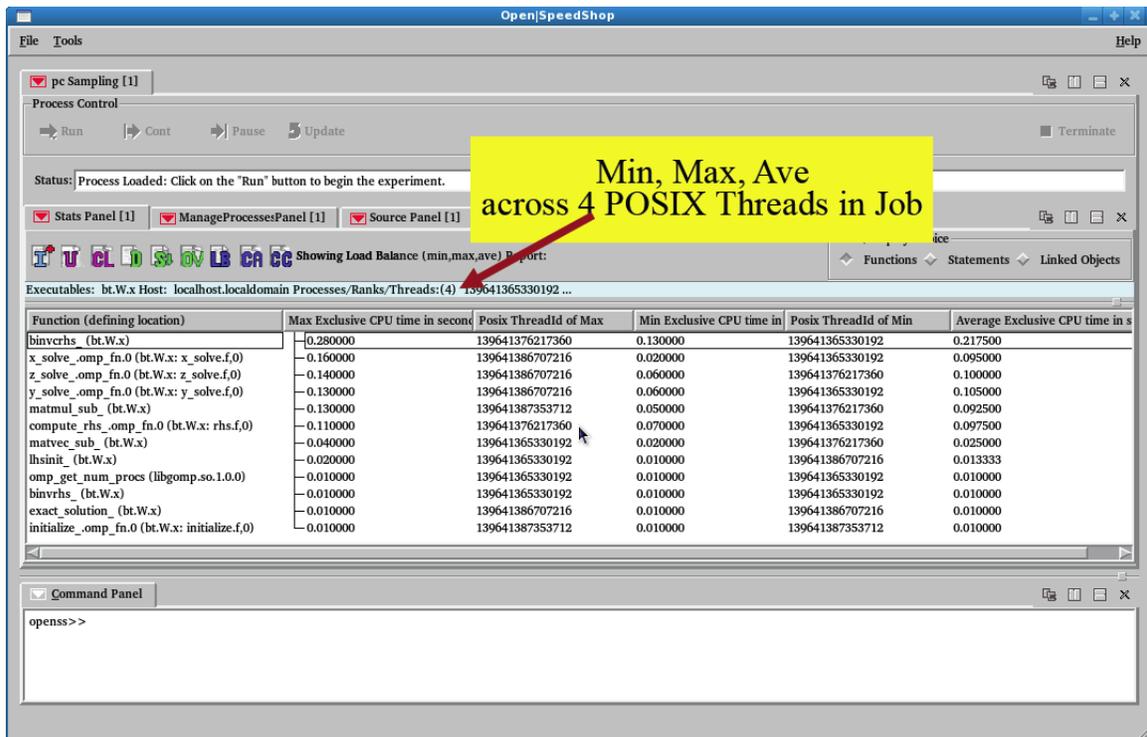
## 8 Threading Analysis Section

We just did an experiment that uses MPI but we can do a similar analysis on applications that use threads. To analyze a threaded application first, we can run the pcsamp experiment to get an overview, then look at the load balance view to detect if there are any widely varying values and finally do cluster analysis to find any outliers.

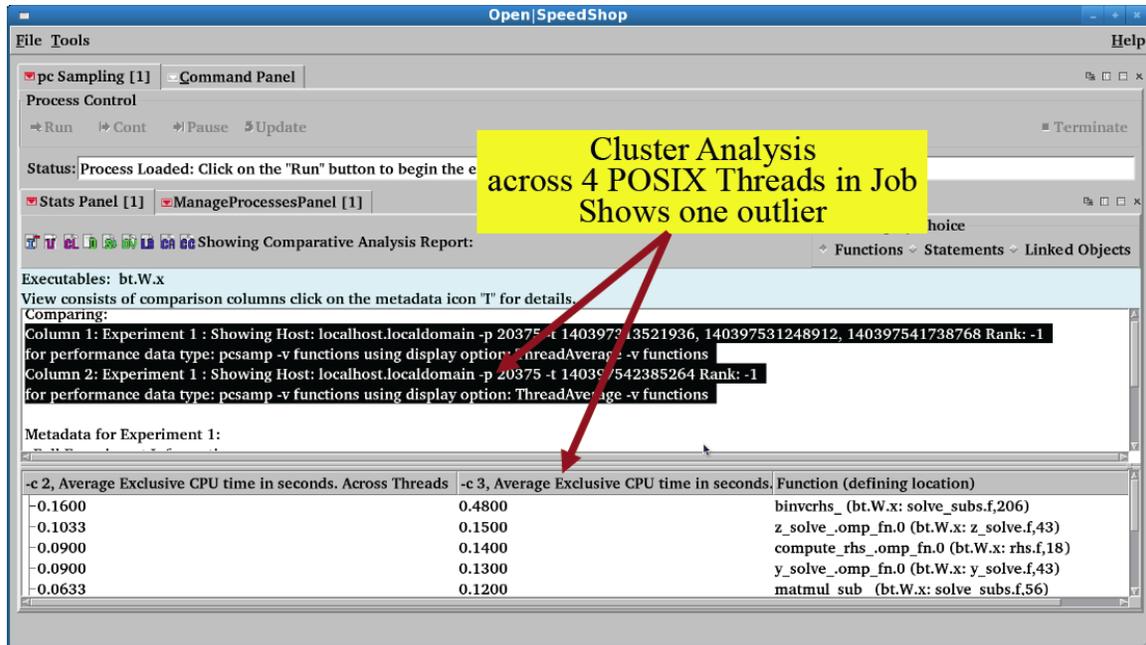
The image below shows the default view for an application with 4 threads, the information displayed is the aggregated total from all threads.



Next we see the load balance view based on functions.



Then we look at a cluster analysis view based on functions.



## 8.1 Threading Specific Experiment (pthreads)

An experiment specific to tracking POSIX thread function calls and analyzing those calls is also available in O|SS. The experiment is called pthreads and it traces several POSIX thread related functions. Like all the other tracing experiments, number of calls, time spent in each function, the call paths to each POSIX thread function, and an event-by-event trace is available. Load balance and cluster analysis features are also available.

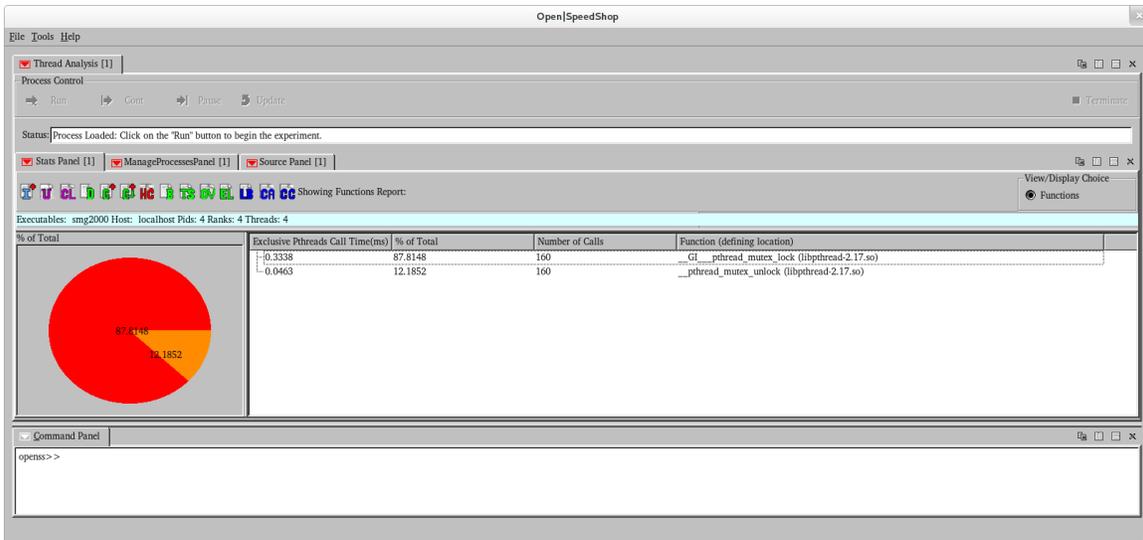
### 8.1.1 Threading Specific (pthreads) experiment performance data gathering

```
osspthread "mpirun -np 4 ./smg2000 -n 15 15 15"
```

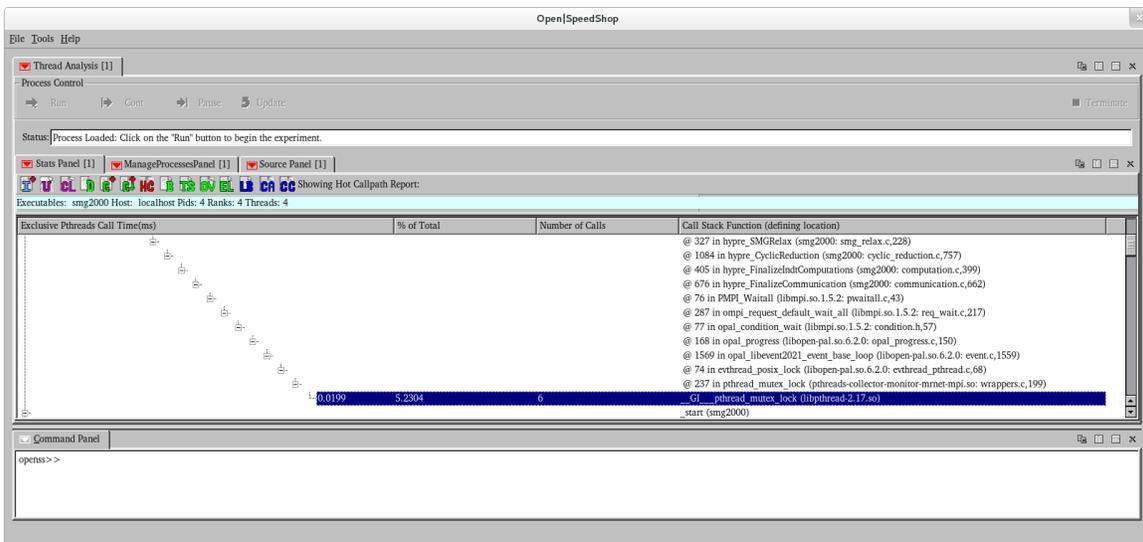
### 8.1.2 Threading Specific (pthreads) experiment performance data viewing with GUI

To launch the GUI on any experiment, use "openss -f <database name>".

Three pthreads experiment views follow. The first is the default pthreads experiment view, which lists the POSIX, thread function routines that were called in the application being monitored, the number of times they were called, and the time spent in each function.



The second view (below) shows the top five, time taking POSIX thread function call paths through the application monitored.



The third view is an event list view, which is a chronological POSIX, thread call list. This view shows each POSIX thread function call in the order they occurred: showing the rank and thread the call originated from, the time spent in the POSIX thread function call event, and the percentage of the total time that represents.





```

77.2600 2.4627 sbrk (libc-2.12.so: sbrk.c,35) 0.2000 1.9189 CalcMonotonicQRegionForElems(Domain&, int,
double*, double) (lulesh2.0: lulesh.cc,1792)
41.7800 1.3317 main (lulesh2.0: lulesh.cc,2690)
34.1000 1.0869 OMPT_THREAD_WAIT_BARRIER (pcsamp-collector-monitor-mrnet-mpi.so:
collector.c,501) 0.6000 0.9754 __psmi_poll_internal (libpsm_infinipath.so.1.14)
25.2300 0.8042 _IO_default_xsputn_internal (libc-2.12.so: genops.c,452)

```

What does the OMPT interface usage in OJSS allow the users of OJSS to do? OJSS applies the OMPT API blame callbacks for `ompt_event_thread_idle`, `ompt_event_thread_barrier` and `ompt_event_thread_wait_barrier` to samples taken in the OpenMP library that otherwise would be shown as `__kmp_barrier`, `__kmp_wait_sleep`, etc. in the Intel `libiomp5` library. We use the `libiomp5` library with the OMPT API enabled at runtime to do this for all OpenMP codes run with the `pcsamp`, `usertime` and hardware counter based experiments. The user can then see the sample time per thread for `idle`, `barrier`, and `wait_barrier`. The user can also use the `loadbalance` metric to see the min, max, average of these blame events or use the `expcompare` across all threads to see individual metrics in comparison to each other.

With respect to the barrier symbols, samples taken when a thread is waiting at a barrier are inclusive to total barrier time. I.e. adding `barrier` and `wait_barrier` metrics is equal to the total barrier time.

Essentially these blame metrics as used in the OJSS sampling experiments inform the user the time a thread is idle and the time spent at a barrier (including waiting at a barrier).

Here is a debug sequence to show that OpenMP thread 2 enters a barrier and then enters a `wait_barrier`. Any sample taken before `wait_barrier_end` should be blamed on the `wait_barrier` (it is in OJSS, however OJSS did not blame any samples that may have been just part of the barrier time itself).

```

[2] CBTF_ompt_cb_barrier_begin parallelID:17 taskID:131 begin time:1460390626061260754
[2] CBTF_ompt_cb_wait_barrier_begin parallelID:17 taskID:131
[2] CBTF_ompt_cb_wait_barrier_end: parallelID:17 taskID:131
[2] CBTF_ompt_cb_barrier_end: parallelID:17 taskID:131 total barrier time:264734
[2] CBTF_ompt_cb_barrier_begin parallelID:17 taskID:131 begin time:1460390626061325546
[2] CBTF_ompt_cb_barrier_end: parallelID:17 taskID:131 total barrier time:269096

```

Here is the case where no `wait_barrier` happened while the OpenMP thread 2 was at a barrier.

Any sample taken in this condition should be blamed on just the barrier.

```

[2] CBTF_ompt_cb_barrier_begin parallelID:15 taskID:117 begin time:1460390626061038145
[2] CBTF_ompt_cb_barrier_end: parallelID:15 taskID:117 total barrier time:196470

```

If one adds `barrier` and `wait_barrier` we would have total samples taken at a barrier and `wait_barrier` is just the samples within that barrier when there is a `thread_barrier_wait` condition.

Here is an example from lulesh sequential OpenMP case:

```
openss>>expview -f OMPT* -m time
Exclusive Function (defining location)
CPU time
  in
seconds.
1.460000 OMPT_THREAD_IDLE (pcsamp-collector-monitor-mrnet.so: collector.c,99)
0.470000 OMPT_THREAD_WAIT_BARRIER (pcsamp-collector-monitor-mrnet.so: collector.c,129)
0.020000 OMPT_THREAD_BARRIER (pcsamp-collector-monitor-mrnet.so: collector.c,113)

openss>>expcompare -f OMPT* -m time -t0:4

-t 0, -t 2, -t 3, -t 4, Function (defining location)
Exclusive Exclusive Exclusive Exclusive
CPU time CPU time CPU time CPU time
  in   in   in   in
seconds. seconds. seconds. seconds.
0.360000 0.030000 0.070000 0.010000 OMPT_THREAD_WAIT_BARRIER (pcsamp-collector-
monitor-mrnet.so: collector.c,129)
0.000000 0.500000 0.400000 0.560000 OMPT_THREAD_IDLE (pcsamp-collector-monitor-
mrnet.so: collector.c,99)
0.000000 0.000000 0.010000 0.010000 OMPT_THREAD_BARRIER (pcsamp-collector-monitor-
mrnet.so: collector.c,113)
openss>>expview -f OMPT* -m time -t3

Exclusive Function (defining location)
CPU time
  in
seconds.
0.400000 OMPT_THREAD_IDLE (pcsamp-collector-monitor-mrnet.so: collector.c,99)
0.070000 OMPT_THREAD_WAIT_BARRIER (pcsamp-collector-monitor-mrnet.so: collector.c,129)
0.010000 OMPT_THREAD_BARRIER (pcsamp-collector-monitor-mrnet.so: collector.c,113)
```

What this shows the user is that most of the barrier samples were taken when the thread were waiting at the barrier. The `OMPT_THREAD_BARRIER` and `OMPT_THREAD_WAIT_BARRIER` symbols replace samples that would have appeared as `__kmp_barrier` (or possibly `__kmp_join_barrier`) in the latest `libiomp5`. The `OMPT_THREAD_IDLE` samples replace `__kmp_wait_sleep` in the real `libiomp5`.

The above applies to `pcsamp`, `usertime`, and the `hwc*` collectors. Essentially telling the user the time a thread is idle and the time spent at a barrier (including waiting at a barrier).

One could infer from the example above with the addition of `OMPT_THREAD_BARRIER` that most of the barrier time for thread 2 was spent waiting at the barrier.

Using the `usertime` experiment on an OpenMP application can help to pinpoint where in the source the wait barrier time is coming from. For example:

```
openss>>expview
Exclusive Inclusive % of Function (defining location)
```

CPU time in seconds.	CPU time in Exclusive seconds.	Total CPU Time	
<b>23.200000</b>	<b>23.200000</b>	<b>38.648263</b>	<b>OMPT_THREAD_IDLE (usertime-collector-monitor-mrnet.so: collector.c,122)</b>
13.142857	13.142857	21.894336	MAIN__omp_fn.2 (stress_omp: stress_omp.f,179)
12.885714	12.885714	21.465969	MAIN__omp_fn.5 (stress_omp: stress_omp.f,227)
<b>4.742857</b>	<b>4.742857</b>	<b>7.901000</b>	<b>OMPT_THREAD_WAIT_BARRIER (usertime-collector-monitor-mrnet.so: collector.c,150)</b>
2.000000	11.771428	3.331747	MAIN__ (stress_omp: stress_omp.f,1)
1.257143	1.257143	2.094241	__kernel_cosf (libm-2.12.so: k_cosf.c,45)
1.085714	1.085714	1.808663	__ieee754_rem_pio2f (libm-2.12.so: e_rem_pio2f.c,108)

Here we see the call path that points to the source lines that result in the thread waiting in the barrier.

```

openss>>expview -vcalltrees,fullstack -f OMPT_THREAD_WAIT_BARRIER usertime1

Exclusive Inclusive % of Call Stack Function (defining location)
CPU time CPU time Total
in in Exclusive
seconds. seconds. CPU Time
_start (stress_omp)
> @ 556 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
>> __libc_start_main (libc-2.12.so)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> main (stress_omp)
>>>>> @ 227 in MAIN__ (stress_omp: stress_omp.f,1)
>>>>>> @ 557 in __kmp_api_GOMP_parallel_end_10_alias (libiomp5.so: kmp_gsupport.c,490)
>>>>>>> @ 2395 in __kmp_join_call (libiomp5.so: kmp_runtime.c,2325)
>>>>>>>> @ 7114 in __kmp_internal_join (libiomp5.so: kmp_runtime.c,7093)
>>>>>>>>> @ 1458 in __kmp_join_barrier(int) (libiomp5.so: kmp_barrier.cpp,1371)
1.742857 1.742857 2.903379 >>>>>>>>>> @ 150 in OMPT_THREAD_WAIT_BARRIER (usertime-collector-monitor-
mnet.so: collector.c,150)

```

The significance of these changes is that we now have the following support in O|SS:

- Added idle, barrier, and wait\_barrier blame support to all sampling collectors
- Improved naming (OMPT\_THREAD\_IDLE, OMPT\_THREAD\_BARRIER, OMPT\_THREAD\_WAIT\_BARRIER) for sample call site.
- Support for building from Intel's libiomp tree to keep us up to data with latest official sources. Should also work with LLVM OpenMP.
- Updated runtime codes to allow running OMPT with gnu/gomp binaries (which means libgomp is replaced with libiomp5 since gomp does not support OMPT directly).
- We also note that the OMPT aspect of OSS works with gcc or g++ generated OpenMP code (and likely clang). No matter what compiler was used to generate the OpenMP code, the Intel libiomp5.so runtime is used for its OMPT API.

With the current OMPT support, there are no distinctions made specific to sub categorization of the idle time or the wait\_barrier time.

Looking at the code in question as found in kmp\_runtime.c, \_\_kmp\_launch\_thread has a while loop that set the ompt state to ompt\_state\_idle, calls \_\_kmp\_fork\_barrier, then sets the ompt state to a default value of ompt\_state\_overhead. If \_\_kmp\_wait\_sleep is called while ompt state is ompt\_state\_idle when kmp\_wait\_sleep is called, then the OMPT API considers the thread "idle". In that case, would you consider this "OMP\_thread\_idle" to be "OMP\_thread\_fork\_wait"?

The other case where `_kmp_wait_sleep` is entered, with an OMPT state of `ompt_state_wait_barrier`, that OMPT state is managed by the `_kmp_barrier` and code `_kmp_join_barrier` in `kmp_runtime.c`. The wait and join barrier are both using `ompt_state_wait_barrier` as coded by the OMPT interface and therefore those are combined in what O|SS reports for `OMP_thread_wait_barrier` in the `pcsamp` example, actually **OMPT\_THREAD\_WAIT\_BARRIER**.

### 8.2.2 Open|SpeedShop OpenMP specific profiling experiment (omptp)

The second form is a separate experiment (`omptp`), which is an OpenMP specific profiling experiment.

## 8.3 Hybrid (openMP and MPI) Performance Analysis

For this tutorial example, we have run O|SS convenience script on the NPB-MZ BT program and created a database file that has 4 ranks each of which has 4 underlying openMP threads.

What this example intends to show is that you can look at hybrid performance first at the MPI level and then can look under the MPI rank to see how the threads are performing. At the MPI level you can see load balance and outliers, then focus on a rank and look at load balance and outliers for the underlying threads. Within a terminal window we enter: `o|ss`

```
o|ss -f bt-mz.B.4-pcsamp-1.o|ss
```

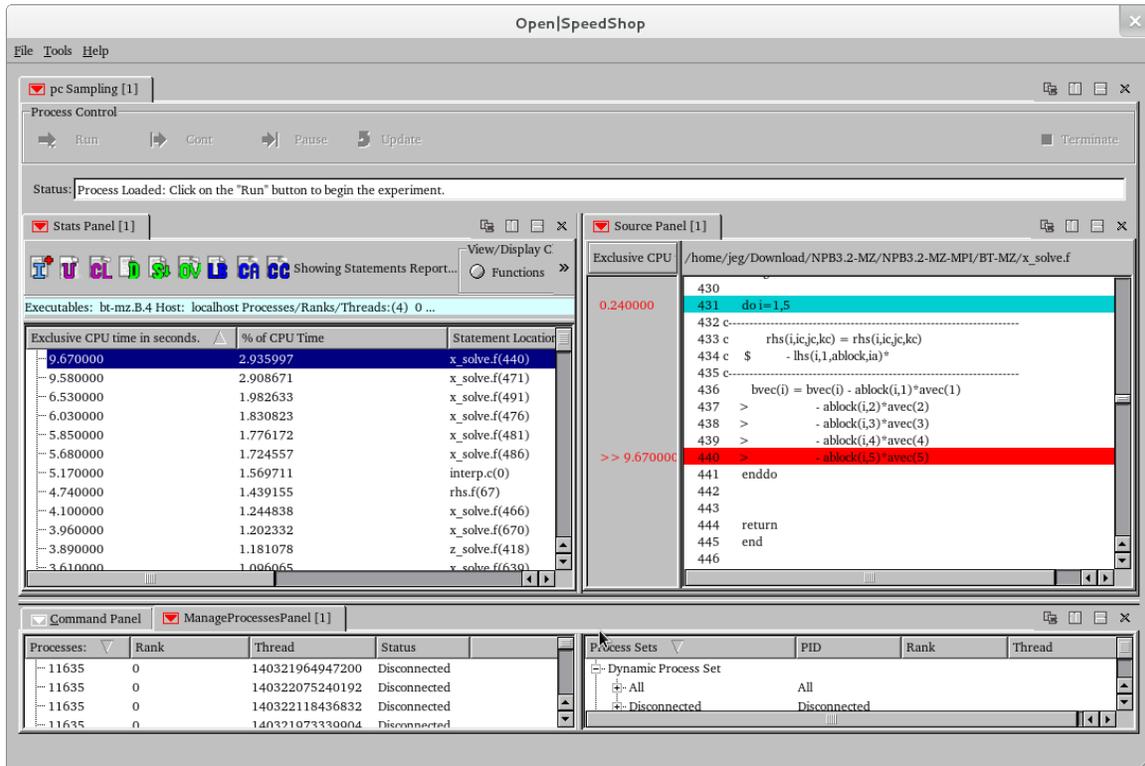
to bring up the O|SS GUI.

In the GUI view below, we display the aggregated results for the application at the statement level granularity. When the default view first comes up the view is at the function level granularity. To switch to the statement level, select the Statements button in the View/Display Choice section on the right hand side of the Stats Panel display and then click the “D” icon for default view. This will switch the Stats Panel view to statement level granularity.

Now the Stats Panel is displaying the statements that took the most time in the application run. For this execution of BT, the statement at line 440 took the most time. By double clicking on the statement, O|SS focuses on the source for that line of the application source and highlights that line.

In the view below, we moved the ManageProcess panel tab to the lower panel and split the upper panel using the vertical splitter icon on the far right side of the original upper panel.

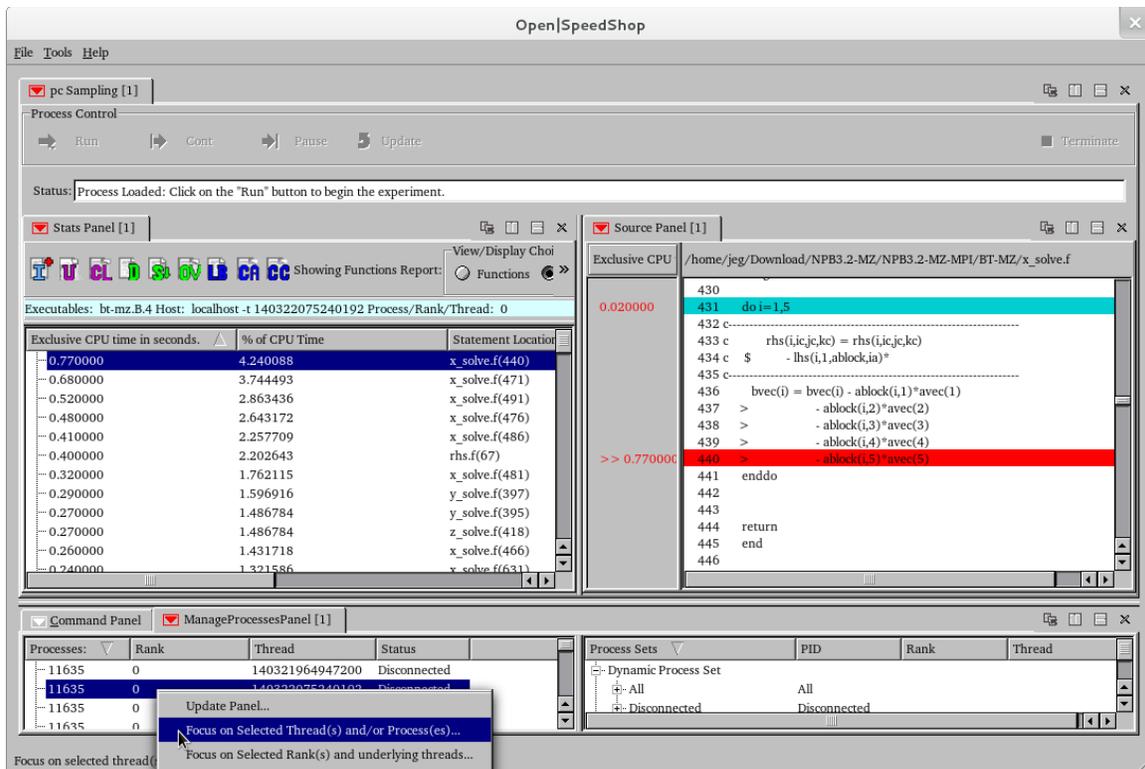
Note: Left mouse down and hold on the panel tab then slide the panel you want to move to another location on the OJSS GUI or off onto other parts of your display.



### 8.3.1 Focus on individual Rank to get Load Balance for Underlying Threads

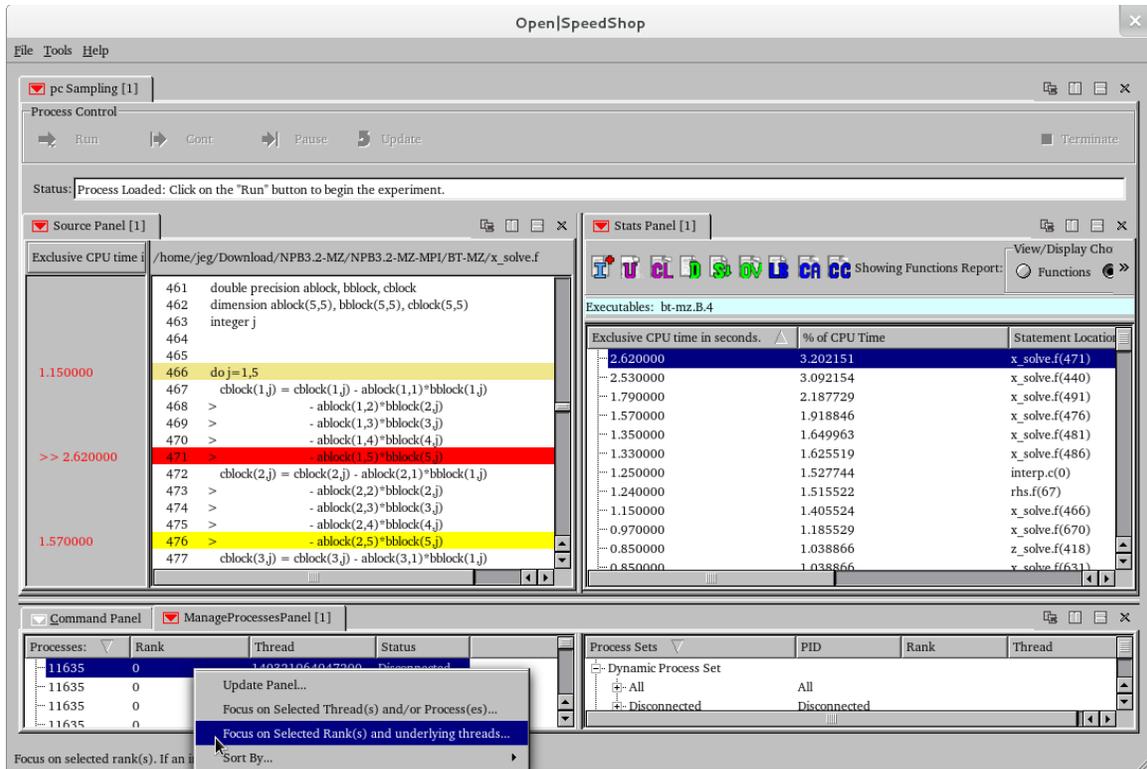
In the next view (below) we used the ManageProcess panel to highlight one rank and an individual thread within the rank to show only that threads performance data in the Stats Panel view.

Note: Use the focus on threads and processes Manage Process panel option to focus on individual threads within a rank. Right mouse button down on the Manage Process panel tab to see the options.



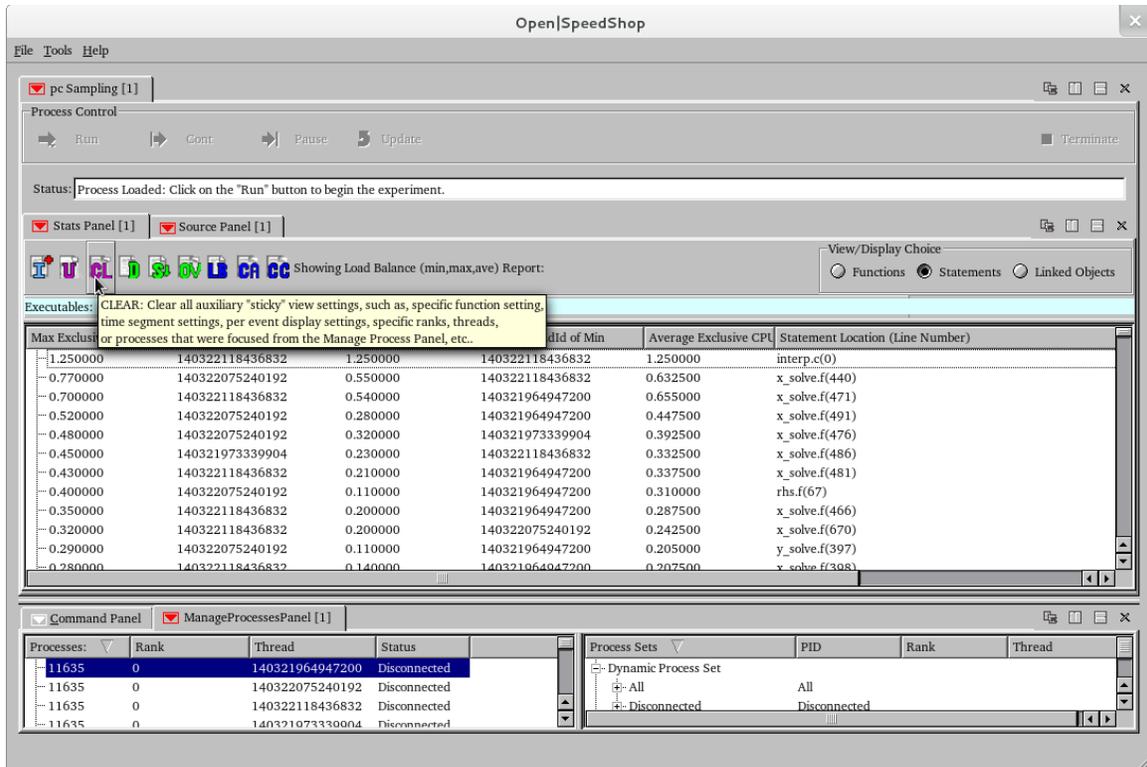
In the next GUI view, we used the ManageProcess panel to highlight one rank to show the performance data from all the threads that are executed under that particular rank in order to see only that performance data in the Stats Panel view.

Note: Use the "focus on selected rank and underlying threads" Manage Process panel option to focus on all the threads within a rank. Right mouse button down on the Manage Process panel tab to see the options.

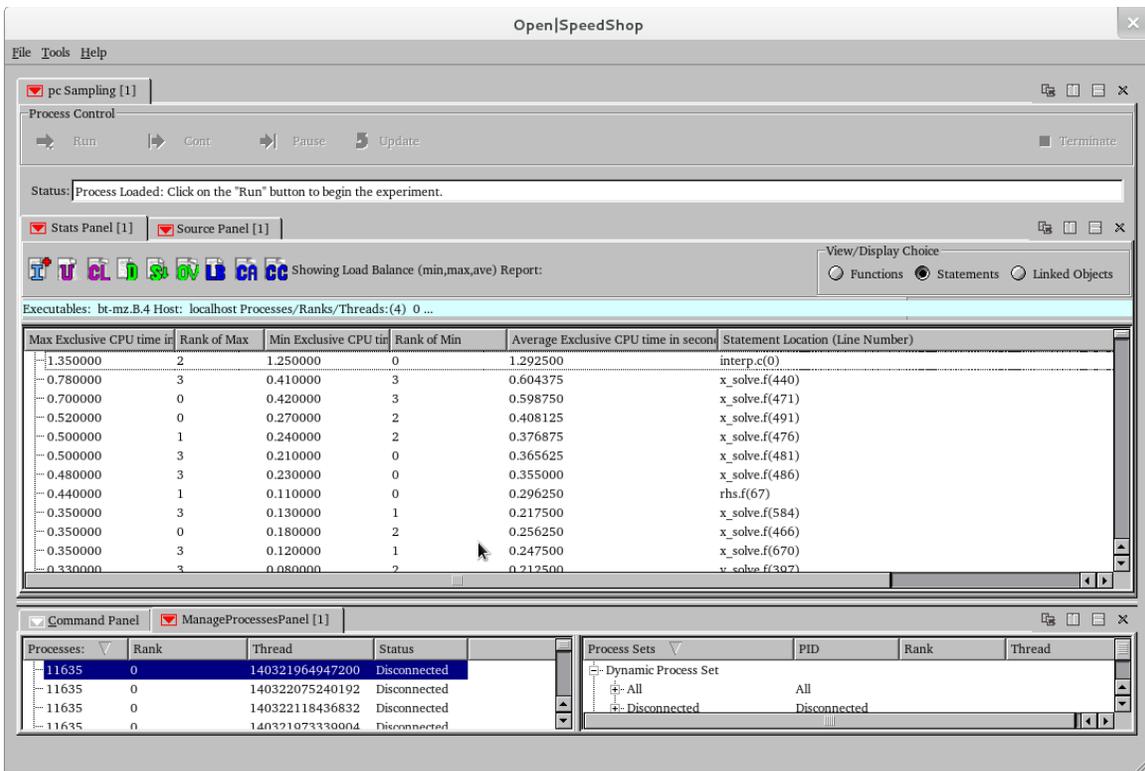


### 8.3.2 Clearing Focus on individual Rank to get bank to default behavior

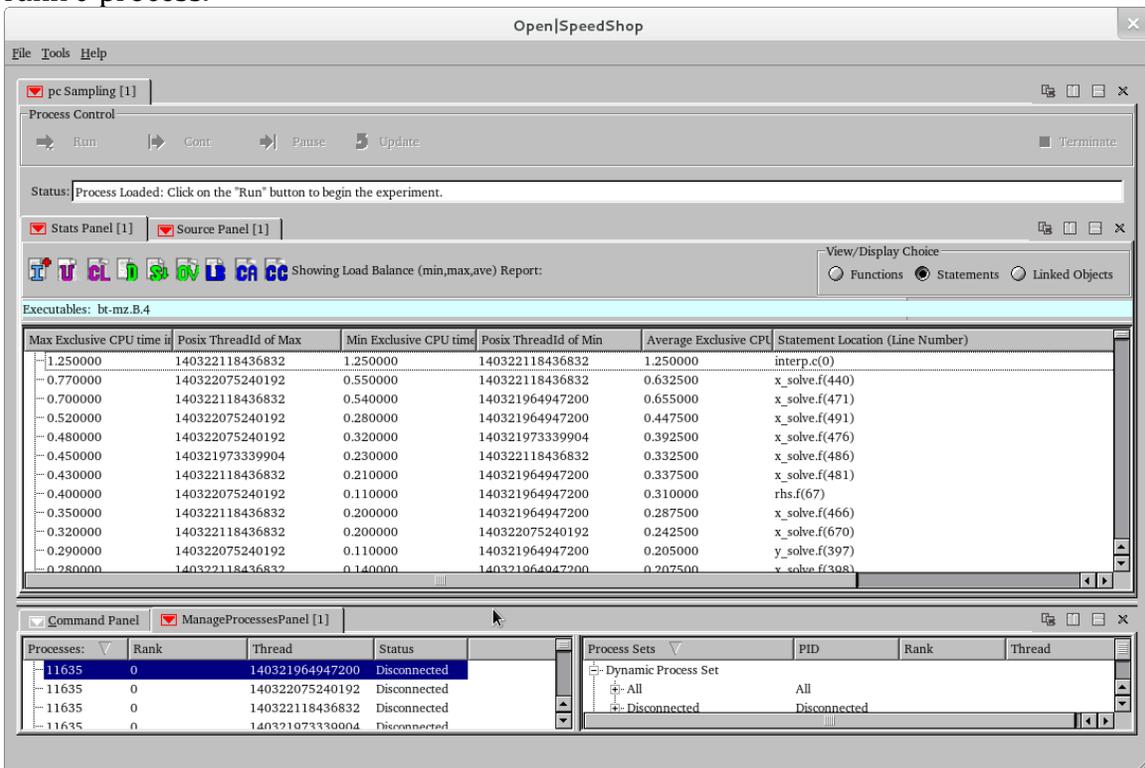
Note: Once you focus on individual or groups of ranks, e.g. venturing away from the default aggregated views, then you need to use the "CL" clear auxiliary setting icon to clear away all the optional selections and get back to looking at the aggregated results again.



After clearing the specific rank and/or thread selections, we can click the "LB" load balance icon and O|SS will display the min, max, average values across all the ranks in the hybrid code. This helps decide if there is imbalance across the ranks of the hybrid application. We can focus on individual ranks to see the balance across the openMP threads that are in an individual rank (next example image).



Here we used the Manage Process panel "Focus on selected rank and underlying threads" menu options to view the load balance across the 4 openMP threads for the rank 0 process.



Please also explore the various options offered via a panel's pull down menu. Clicking on a colored downward-facing arrow or using the Stats Panel icons can access further options. Red icons represent view options, such as updating the data or clearing the view options. The "green" icons correspond to different possible views of the performance data. The "dark blue" icons correspond to analysis options while the "light blue" icon corresponds to information about the experiment. There is context sensitive text that is shown when you hover over the icons.

## 9 GPU Performance Analysis

### 9.1 NVIDIA CUDA Analysis Section

The OJSS version with CBTF collection mechanisms supports tracing CUDA events in a NVIDIA CUDA based application. An event-by-event list of CUDA events and the event arguments are gathered and displayed.

#### 9.1.1 NVIDIA CUDA Tracing (cuda) experiment performance data gathering

To run the NVIDIA CUDA experiment, use the osscuda convenience script and specify the CUDA application as an argument. Here is the general format of the osscuda convenience script that is used to gather the NVIDIA CUDA performance information.

```
osscuda "how you run your application normally"
```

In this example, the osscuda script will run the experiment by running the GEMM application and will create an OJSS database file with the results of the experiment. Viewing of the performance information can be done with the GUI or CLI. A default CLI text based report is displayed at the end of the application run.

```
osscuda "mpirun -np 2 -ppn 1 -hosts ccn001,ccn002 ./GEMM"  
[openss]: cuda counting all instructions for CPU and GPU.  
[openss]: cuda using default periodic sampling rate (10 ms).  
[openss]: cuda configuration: "interval=1000000,PAPI_TOT_INS,inst_executed"  
Creating topology file for slurm frontend node ccn001 for SLURM_JOB_ID 131  
Generated topology file: ./cbtfAutoTopology  
Running cuda collector.  
Program: mpirun -np 2 -ppn 1 -hosts ccn001,ccn002 ./GEMM  
Number of mrnet backends: 2  
Topology file used: ./cbtfAutoTopology  
executing mpi program: mpirun -np 2 -ppn 1 -hosts ccn001,ccn002 cbtfrun --mpi --mrnet -c  
cuda ./GEMM  
MPI Task 0/1 starting...  
MPI Task 1/1 starting...  
Chose device: name='Tesla K40c' index=0  
Running single precision test
```

```

Chose device: name='Tesla K40c' index=0
Running single precision test
Running double precision test
Running double precision test
test atts units median mean stddev min max
DGEMM-N(max) 128 GFlops 57.5899 57.5899 0.259358 57.3306 57.8493
DGEMM-N(mean) 128 GFlops 56.9609 56.9609 0.0587321 56.9022 57.0196
DGEMM-N(median) 128 GFlops 57.3772 57.3772 0.395575 56.9816 57.7728
DGEMM-N(min) 128 GFlops 54.036 54.036 1.59142 52.4445 55.6274
...
...
...
Extreme outliers (>3.0 IQR from 1st/3rd quartile):
None.
default view for ./GEMM-cuda-3.openss

[openss]: The restored experiment identifier is: -x 1
Performance data spans 0.452563 ms from 2016/11/09 22:35:07 to 2016/11/09 22:35:07

Exclusive % of Exclusive Function (defining location)
Time (ms) Total Count
Exclusive
Time
9.861216 52.062327 200 void RunTest<float>(std::string, ResultDatabase&, OptionParser&)
(GEMM: GEMM.cpp,156)
9.079958 47.937673 200 void RunTest<double>(std::string, ResultDatabase&, OptionParser&)
(GEMM: GEMM.cpp,156)

```

### 9.1.2 NVIDIA CUDA Tracing (cuda) experiment performance data viewing with GUI

This section shows the default view for the NVIDIA CUDA experiment for the QTC application. Use the following command to open the GUI to see the GEMM CUDA experiment performance information.

To launch the new beta, cuda focused, GUI on any experiment, use:

```

"openss-gui -f <database name>".

```

NOTE: This is a different GUI from the existing O|SS GUI. It is being developed initially focused on providing views for the NVIDIA cuda experiment. Use openss-gui instead of openss to invoke this GUI.

```

openss-gui -f GEMM-cuda-0.openss

```

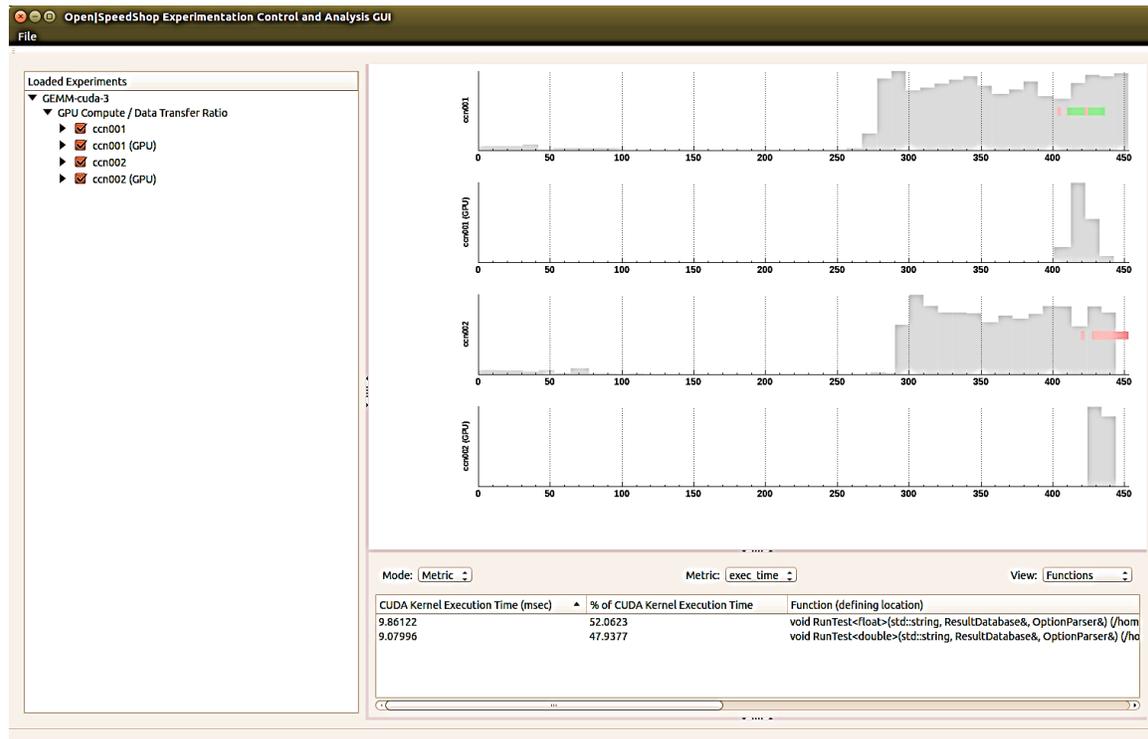
The view below is a default for cuda experiment. A time line tracking the cuda kernel executions are shown with a green color and data transfers to the GPU device and the CPU are shown with a red color. This is too give insight into the relative cost of the transfers versus the actual time spent executing in the kernel. The shaded areas in the timeline represent performance information from the execution

of the application in the GPU and in the CPU. So, one can infer if the GPU and CPU are being fully utilized.

The left panel, labeled “Loaded Experiments” is the area where one will be able to eventually select what process, thread, or rank performance information will be displayed in the timeline and metric value display panel. Currently, it shows information about the application and hosts (processes, threads, ranks) that are involved in this GUI display.

The metric panel (lower right display panel) is where the text based performance information is displayed. There are options based on metric type that control the data being presented in this panel.

A source view panel is also available by opening a GUI pane below the metric panel.



### 9.1.3 NVIDIA CUDA Tracing (cuda) experiment performance data viewing with CLI

To launch the CLI on any experiment, use “`openss -cli -f <database name>`”.

The OJSS CLI will report CUDA kernel execution, CUDA data transfer, and CPU/GPU hardware performance counter data gathered by the 'cuda' collector.

The type of data displayed can be controlled through the '-v' options:

Exec    CUDA kernel executions (this is the default)  
Xfer    CUDA data transfers  
HWPC    CPU/GPU hardware performance counters

The form of the displayed information is controlled through additional '-v' options. For '-v Exec' and '-v Xfer' these additional '-v' options are:

ButterFly            Produces a report summarizing the calls to and from the one or more functions specified by the '-f <function\_list>' option. Calling functions will be listed before the named function, and called functions afterwards, by default, unless 'TraceBacks' is specified to reverse this ordering.

CallTree[s]            Produces a calling stack report presented in calling tree order, from the executable's start toward the measurement locations.

(DSO|LinkedObject)[s]    Produces a summary report by linked object.

FullStack[s]            Causes the report to include the full call stack for each measurement location when added to either 'CallTree' or 'TraceBack'. Redundant call stack frames are suppressed by default if this option isn't specified.

Function[s]            Produces a summary report by function. This is the default.

Loop[s]                Produces a summary report by loop.

Statement[s]            Produces a summary report by statement.

Summary                Causes the report to include an additional line of output at the end that summarizes the information in each column. Does not apply to 'ButterFly' or 'Trace'.

SummaryOnly            Causes the report to ONLY include the line of output generated by 'Summary'.

Trace                  Produces a report of each individual CUDA kernel execution or data transfer, sorted in

ascending order of the event's start time.

TraceBack[s] Produces a calling stack report presented in traceback order, from the measurement locations toward the executable's start.

Except for the '-v Trace' option, the report will be sorted in descending order of the values in the leftmost column. Multiple '-v' values can be delimited with commas. E.g. '-v Exec,Trace'.

Finally, the columns included in the report can be controlled using the '-m' option. More than one column may be specified in a comma-delimited list. And when '-m' is used, ONLY those columns specified are reported, in order they are given.

The following '-m' options are available for '-v [Exec|Xfer]':

[%][exclusive_]count[s]	Exclusive number of events
[%]inclusive_count[s]	Inclusive number of events
[%][exclusive_]time[s]	Exclusive time in the event
[%]inclusive_time[s]	Inclusive time in the event
min[imum]	Minimum time in the event
max[imum]	Maximum time in the event
avg[erage]	Average time in the event
stddev	Standard deviation of time in the event
ThreadMin	Minimum accumulated time for a process
ThreadMinIndex	Process ID of the 'ThreadMin' process
ThreadMax	Maximum accumulated time for a process
ThreadMaxIndex	Process ID of the 'ThreadMax process'
ThreadAverage	Average accumulated time for a process
LoadBalance	Equivalent to 'ThreadMax, ThreadMaxIndex, ThreadMin, ThreadMinIndex, ThreadAverage'.

The following '-m' options are only available for '-v [Exec|Xfer],Trace':

(start|stop)[\_time] Start or stop time for the event

The following '-m' options are only available for '-v Exec,Trace':

block	Dimensions of each block
cache	Cache preference used
dsm	Total amount (in bytes) of dynamic shared memory reserved
grid	Dimensions of the grid
lm	Total amount (in bytes) of local memory reserved
rpt	Registers required for each thread

ssm Total amount (in bytes) of static shared reserved

The following '-m' options are only available for '-v Xfer,Trace':

size Number of bytes being transferred  
kind Kind of data transfer performed  
src Kind of memory from which the data transfer was performed  
dest Kind of memory to which the data transfer was performed  
async Was the data transfer asynchronous?

The default columns used for various '-v' combinations are:

-v Exec,Trace -m start,time,%time,grid,block  
-v Xfer,Trace -m start,time,%time,size,kind  
-v (Exec|Xfer),Butterfly -m inclusive\_time,%inclusive\_time  
-v (Exec|Xfer)[,<all-other>] -m time,%time,count

The '-v HWPC' view works differently in that it only displays the sampled CPU/GPU hardware performance counters as a function of time. I.e. it does not display data as a function of source code constructs. Thus only the '-v Summary' and '-v SummaryOnly' options apply.

It also interprets the positive integer added to the end of the keyword 'cuda' differently. Instead of being the maximum number of reported items it specifies the fixed sampling interval (in ms) at which the data should be resampled before display. The default value (0) is given the special meaning that the original sampling interval should be used instead.

Examples:

```
expView cuda  
expView -v Xfer,CallTrees,FullStack cuda10 -m min,max,count  
expView -v HWPC,Summary cuda33
```

See also:

expView

Here are some CLI views of the output from the osscuda experiment. These views show results of a cuda experiment on the CUDA application GEMM on the Pleiades SGI platform at NASA.

```
pfe27-433>openss -cli -f GEMM-cuda-4.openss  
openss>>[openss]: The restored experiment identifier is: -x 1  
openss>>expview
```

```
Exclusive % of Exclusive Function (defining location)  
Time (ms) Total Count
```

```

Exclusive
Time
14.810702 52.042113 300 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
13.648369 47.957887 300 void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
opnss>>expstatus

```

```

Experiment definition
{ # ExpId is 1, Status is Terminated, Saved database is GEMM-cuda-4.opnss
  Performance data spans 0.443760 ms from 2016/08/24 10:01:03 to 2016/08/24 10:01:03
  (none)
  Executables Involved:
  (none)
  Currently Specified Components:
  -h maia29 -p 43727 -t 0 -r 0
  -h maia30 -p 27136 -t 0 -r 1
  -h maia31 -p 80595 -t 0 -r 2
  Previously Used Data Collectors:
  cuda
  Metrics:
  cuda::count_exclusive_details
  cuda::exec_exclusive_details
  cuda::exec_inclusive_details
  cuda::exec_time
  cuda::xfer_exclusive_details
  cuda::xfer_inclusive_details
  cuda::xfer_time
  Parameter Values:
  Available Views:
  cuda
}

```

```
opnss>>expview -vExec
```

```

Exclusive  % of Exclusive Function (defining location)
Time (ms)  Total  Count
Exclusive
Time
14.810702 52.042113 300 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
13.648369 47.957887 300 void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
opnss>>expview -vXfer

```

```

Exclusive  % of Exclusive Function (defining location)
Time (ms)  Total  Count
Exclusive
Time
1.774178 75.232917 69 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
0.584069 24.767083 69 void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
opnss>>expview -v trace, Xfer

```

```

Start Time (d:h:m:s)  Exclusive  % of  Size  Kind Call Stack Function (defining location)
Time (ms)  Total
Exclusive
Time
2016/08/24 10:01:03.845 0.001217 0.051606 112 HostToDevice >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.850 0.027392 1.161541 262144 HostToDevice >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.850 0.027553 1.168368 262144 HostToDevice >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.001217 0.051606 112 HostToDevice >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.027425 1.162940 262144 DeviceToHost >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.852 0.026721 1.133087 262144 DeviceToHost >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)

```

```

2016/08/24 10:01:03.852 0.026753 1.134444 262144 DeviceToHost >>void RunTest<float>(std::string,
ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
.....

openss>>expview -v trace,Exec

Start Time (d:h:m:s)  Exclusive  % of  Grid  Block  Call Stack Function (defining location)
Time (ms)  Total  Dims  Dims
                Exclusive
                Time
2016/08/24 10:01:03.851 0.055585 0.195316 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.048705 0.171141 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.049761 0.174851 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.051617 0.181373 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.051648 0.181482 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.050817 0.178562 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.046496 0.163378 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.851 0.048193 0.169341 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
2016/08/24 10:01:03.852 0.049633 0.174401 4,4,1 16,16,1 >>void RunTest<float>(std::string, ResultDatabase&,
OptionParser&) (GEMM: GEMM.cpp,19)
....
openss>>expview -vcalltrees,fullstack

Exclusive  % of  Exclusive  Call Stack Function (defining location)
Time (ms)  Total  Count
                Exclusive
                Time
                main (GEMM: main.cpp,135)
                > @ 130 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122)
11.818358 41.527561 240 >> @ 240 in void RunTest<float>(std::string, ResultDatabase&, OptionParser&)
(GEMM: GEMM.cpp,19)
                main (GEMM: main.cpp,135)
                > @ 137 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122)
10.894840 38.282486 240 >> @ 240 in void RunTest<double>(std::string, ResultDatabase&, OptionParser&)
(GEMM: GEMM.cpp,19)
                main (GEMM: main.cpp,135)
                > @ 130 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122)
2.992344 10.514553 60 >> @ 231 in void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM:
GEMM.cpp,19)
                main (GEMM: main.cpp,135)
                > @ 137 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122)
2.753529 9.675400 60 >> @ 231 in void RunTest<double>(std::string, ResultDatabase&, OptionParser&)
(GEMM: GEMM.cpp,19)

```

## 10 Memory Analysis Techniques

The O|SS version with CBTF collection mechanisms supports tracing memory allocation and deallocation function calls in user applications. An event-by-event list of memory function call events and the memory function call event arguments are listed. The O|SS experiment name for the memory analysis experiment is “mem”. The high water memory mark is not currently available but is coming in the future.

### 10.1 Memory Analysis Tracing (mem) experiment performance data gathering

To run the memory analysis experiment, use the `ossmem` convenience script and specify the application as an argument. If there are no arguments to the application then no quotes are necessary, but they are placed here for consistency. Using the `sweep3d` application as an example, here the `ossmem` script will apply the memory analysis experiment by running the `sweep3d` application with the O|SS memory trace collector, gather the data and will create an O|SS database file with the results of the experiment. Viewing of the performance information can be done with the GUI or CLI.

```
ossmem "mpirun -np 64 ./sweep3d.mpi"
```

### 10.2 Memory Analysis Tracing (mem) experiment performance data viewing with CLI

To launch the CLI on any experiment, use “`openss -cli -f <database name>`”. The following table describes the fields in the memory experiment default CLI view.

Column Name	Column Definition
Exclusive Mem Call Time	Aggregated total exclusive time spent in the memory function corresponding to this row of data.
% of Total Time	Percentage of exclusive time relative to the total time spent in the memory function corresponding to this row of data.
Number of Calls	Total number of calls to the memory function corresponding to this row of data.
Min Request Count	The number of times minimum bytes allocated or freed occurred during this experiment.
Min Requested Bytes	The minimum number of bytes that were allocated or freed by the corresponding memory function.
Max Request Count	The number of times maximum bytes allocated or freed occurred during this experiment.
Max Requested Bytes	The maximum number of bytes that were allocated or freed by the corresponding memory function.
Total Requested Bytes	The total number of bytes allocated by the corresponding function. Note: this does not subtract the bytes freed. This only totals the allocation function requested bytes.







```

>>_libc_start_main (libc-2.18.so)
>>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
>>>> @ 37 in main (mpiperf: mpiperf.c,22)
>>>>> @ 314 in mp_parse_cmd (mpiperf: mpiperf.c,57)
>>>>>> @ 692 in mp_allreduce (mpiperf: mpiperf.c,686)
>>>>>>>MPI_Allreduce (libmpi.so.1.6.0)
>>>>>>>>ompi_coll_tuned_allreduce_intra_recurseiddoubling (libmpi.so.1.6.0)
16 400000 >>>>>>>>>malloc (libc-2.18.so)

```

Here is where mpiperf.c is calling MPI\_Allreduce which is doing all the mallocs.

```

openss>>lsed -n 680,700p mpiperf.c

```

```

openss>> stop = MPI_Wtime();

```

```

    return stop - start;
}

```

```

double mp_allreduce(int msg_size, int steps, int rank, int size)

```

```

{
    double start, stop;
    int i;

    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();
    for (i=0; i < steps; i++) {
        MPI_Allreduce(sendbuf, recvbuf, msg_size, mp_data_type, mp_op, MPI_COMM_WORLD);
    }
    stop = MPI_Wtime();

```

```

    return stop - start;
}

```

```

double mp_alltoall(int msg_size, int steps, int rank, int size)

```

These are additional important -m options, as these are per call.

See the man pages of these calls for the details on the return values and arguments.

```

void *malloc(size_t size);

```

-m retval shows the address to the returned memory

-m size1 shows the amount of the allocation request.

```

void free(void *ptr);

```

-m ptr shows the address of the free request. matching this to addresses allocated in any

of the malloc calls will show if an allocation is free'd or not. not done in our views.

one would have to dump a complete -v trace to a text file and matchup allocations and frees.

```

void *calloc(size_t nmemb, size_t size);

```

-m retval shows the address to the returned memory

-m size1 = nmemb

-m size2 = size

```
void *realloc(void *ptr, size_t size);
```

-m retval shows the address to the returned memory

-m ptr is the ptr to the existing mem location

-m size1 is the realloc request size

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

-m ptr is the pointer to the memory address

-m size1 is alignment

-m size2 is size

-m retval is the address that would later be passed to free

```
void *memalign(size_t alignment, size_t size);
```

-m retval shows the address to the returned memory

-m size1 is alignment

-m size2 is size

Any call that has a size parameter can use

-m max\_bytescount

-m min\_bytescount

-m max\_bytes

-m min\_bytes

-m tot\_bytes

This will show total allocations done per unique callpath, add -mcounts to see number of times traversed:

```
expview -v calltrees,fullstack -mtot_bytes
```

This will sort on largest to smallest MAX allocations done per unique callpath, add -mcounts to see number of times traversed:

```
expview -v calltrees,fullstack -mmax_bytes
```

This will sort on largest to smallest MIN allocations done per unique callpath, add -mcounts to see number of times traversed:

```
expview -v calltrees,fullstack -mmin_bytes
```

In some cases, the MAX and MIN will match if they are equal for any particular allocation path.

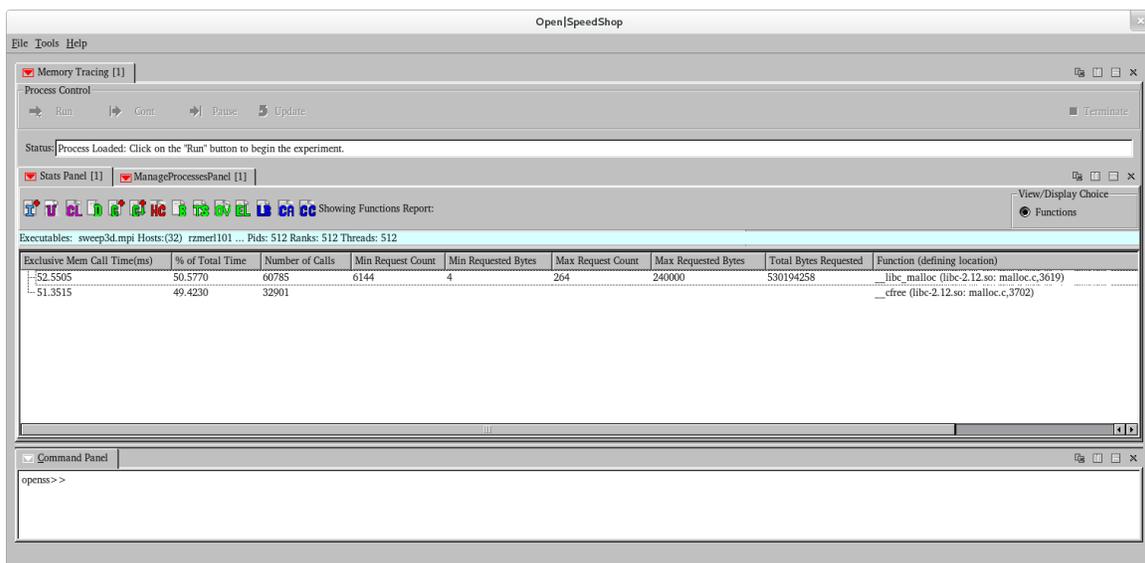
### **10.3 Memory Analysis Tracing (mem) experiment performance data viewing with GUI**

To launch the GUI on any experiment, use “openss -f <database name>”.

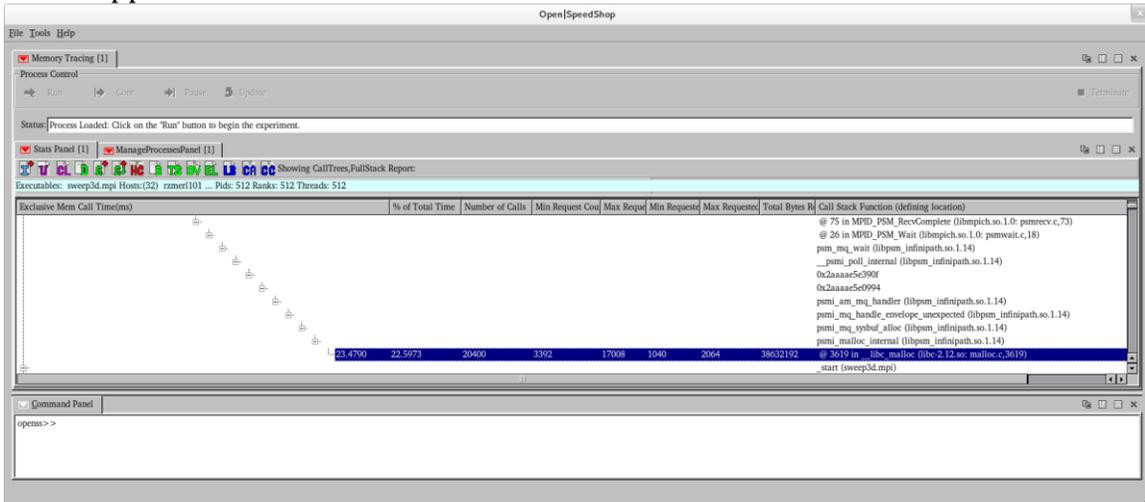
The first GUI view show below is the default view for the mem experiment. It shows the memory functions that were called in the application, how many times they were called, the time spent in each of the memory functions, and the percentage of the overall memory function time was spent in each of the memory functions. This table identifies what each of the columns represent in the default GUI view for the mem experiment.

Column Name	Column Definition
Exclusive Mem Call Time	Aggregated total exclusive time spent in the memory function corresponding to this row of data.
% of Total Time	Percentage of exclusive time relative to the total time spent in the memory function corresponding to this row of data.
Number of Calls	Total number of calls to the memory function corresponding to this row of data.
Min Request Count	The number of times minimum bytes allocated or freed occurred during this experiment.
Min Requested Bytes	The minimum number of bytes that were allocated or freed by the corresponding memory function.
Max Request Count	The number of times maximum bytes allocated or freed occurred during this experiment.
Max Requested Bytes	The maximum number of bytes that were allocated or freed by the corresponding memory function.
Total Requested Bytes	The total number of bytes allocated by the corresponding function. Note: this does not subtract the bytes freed. This only totals the allocation function requested bytes.

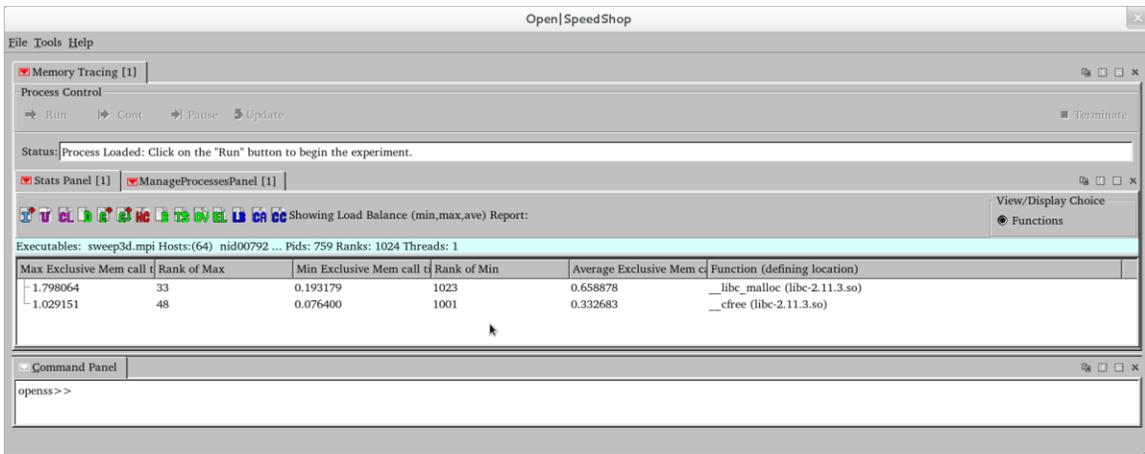
The paths to each memory, through the source, are available through the call path views.



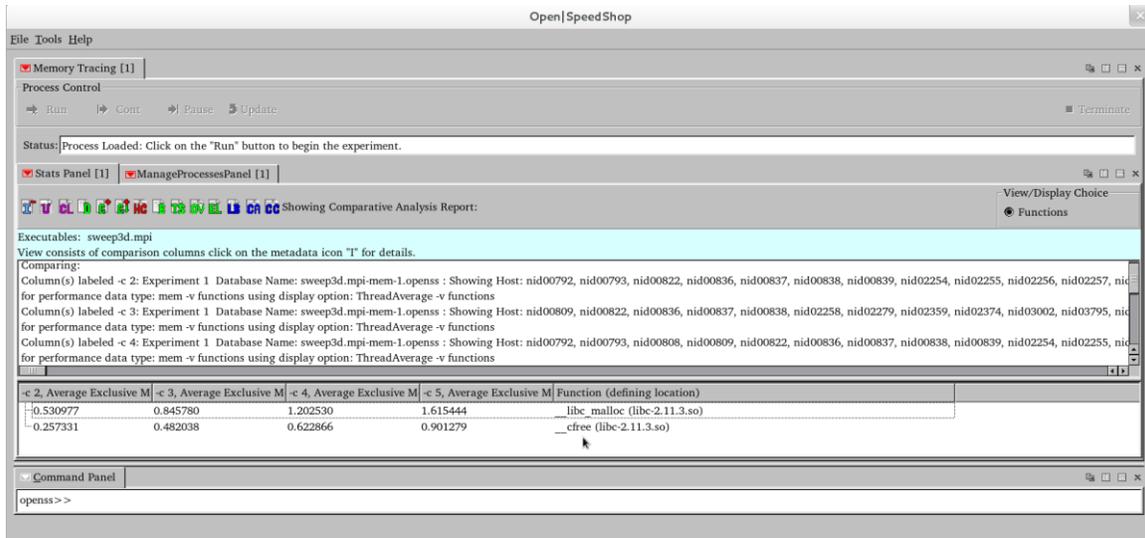
In this (C+ icon) call path view we see the call paths to the memory functions called in this application.



In the view below, one has chosen the “LB” icon and generated the load balance view. This view shows the min, max, and average time across all the ranks in the application. The ranks of the min and max time values are also shown. If there is a significant difference between the min, max, and average time, there may be load imbalance. To identify the ranks, threads, or processes that are acting out of balance, use the cluster analysis feature activated by clicking on the “CA” icon.



In this view, generated by clicking on the “CA” icon, we see that O|SS has determined that there are four unique groups where the aggregate time for the groups differs enough to report this to the user. The columns in the Stats Panel display show the times that are reflective of each of the ranks in the group. The information (I+) icon can be used to view which ranks, etc. are included in each of the cluster groups.



## 11 Advanced Analysis Techniques

Analyzing the results of a single performance experiment can be useful for debugging and tuning your code. But comparing the results of different experiments can show you how the performance of an application has changed. This is useful if you want to track how the performance varies for each new version of an application, or understanding how a different compiler or compiler options can affect the performance of your application. This also allows you to do scalability tests to see how the performance of your application scales with the number of processors. It's also helpful just to see the progress you have made while tuning your code.

O|SS has options to allow you to compare performance data. You can use the Custom Compare Panel (CC icon) in the GUI or the `osscompare` convenience script.

```
> osscompare "db1.openss, db2.openss,..." [options]
```

This will produce a side-by-side comparison listing, you can compare up to 8 databases at once. You can see the `osscompare` man page for more details. Below is an example of comparing two different `pcsamp` experiments on the `smg2000` application.

```
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss"
```

```
[openss]: Legend: -c 2 represents smg2000-pcsamp.openss
```

```
[openss]: Legend: -c 4 represents smg2000--pcsamp---1.openss
```

```
-c 2, Exclusive CPU          -c 4, Exclusive CPU      Function (defining location)
```

```
time in seconds.
```

```
time in seconds.
```

```
3.870000000          3.630000000          hypre_SMGResidual (smg2000:smg_residual.c,152)
```

```
2.610000000          2.860000000          hypre_CyclicReduction (smg2000:cyclic_reduc;on.c,757)
```

```
2.030000000          0.150000000          opal_progress (libopen-pal.so.0.0.0)
```

```
1.330000000          0.100000000          mca_btl_sm_component_progress (libmpi.so.0.0.2)
```

```
0.280000000    0.210000000    hypre_SemiInterp (smg2000: semi_interp.c,126)
0.280000000    0.040000000    mca_pml_ob1_progress (libmpi.so.0.0.2)
```

## 11.1 Comparison Script Argument Description

The O|SS comparison script accepts a number of arguments. This section describes the acceptable options for those individual arguments. For a quick overview see section 14.4 *osscompare: Compare Database Files*. As described above the `osscompare` script accepts at least two and up to eight comma separated database file names, enclosed in quotes as the mandatory argument. By default, the compared metric is the primary metric produced by the experiment. For most experiments, the metric is exclusive time, however the hardware counter experiments use the count of the number of hardware counter overflows as the metric to be compared. These are the default or mandatory arguments to `osscompare`. The following sections describe the arguments for `osscompare` in more detail.

### 11.1.1 `osscompare` metric argument

The `osscompare` metric argument specifies the performance information type that O|SS will use to compare against when looking at each database file in the compare database file list. To find the metric specifications that are legal and produce comparison outputs, one can open one of the database files with the O|SS command line interface (CLI), and list the available metrics.

```
openss -cli -f smg2000-pcsamp.openss
openss>>list -v metrics
pcsamp::percent
pcsamp::threadAverage
pcsamp::threadMax
pcsamp::threadMin
pcsamp::time
```

You can use the output of the `list metrics` command as an argument to the `osscompare` command as shown in the examples below.

```
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss"
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" percent
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" threadMin
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" threadMax
```

Some exceptions do apply. For example, some experiments such as `usertime` and `hwctime` have “details” type metrics output by the `list metrics` CLI command (`list -v metrics`). These will not work as a metric argument to `osscompare`.

For the hardware counter experiments: `hwc` and `hwctime`, you can use the actual PAPI event name in addition to the metric names output from the `list metric` command. The example database file was generated using the `PAPI_TOT_CYC` event.

```
openss -cli -f smg2000-hwc.openss
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>list -v metrics
hwc::overflows
hwc::percent
hwc::threadAverage
hwc::threadMax
hwc::threadMin
```

Here, we show a couple osscompare examples where “hwc::overflows” can be used interchangeably with PAPI\_TOT\_CYC.

```
osscompare "smg2000-hwc.openss,smg2000-hwc-1.openss" hwc::overflows
osscompare "smg2000-hwc.openss,smg2000-hwc-1.openss" PAPI_TOT_CYC
```

Note that for compares involving hwcsamp metric based databases, to compare all of the existing hardware counters from each experiment, use the “allEvents” metric in the osscompare command. That will compare all the events in each of the databases and will ignore the program counter sampling data from each of the databases. The form of the osscompare command to compare all the hardware counter events is as follows:

```
osscompare "smg2000-hwcsamp.openss,smg2000-hwcsamp-1.openss" allEvents
```

### 11.1.2 osscompare rows of output argument

osscompare allows the user to specify how many lines of the comparison output to be output. The argument is optional and

```
"rows=nn" is defined as follows:
"nn" - Number of rows/lines of performance data output.
```

In this example, only ten (10) lines of comparison will be shown when the osscompare command is executed. It will be the most interesting, or top, ten lines.

```
osscompare "smg2000-hwc.openss,smg2000-hwc-1.openss" hwc::overflows rows=10
```

### 11.1.3 osscompare output name argument.

osscompare allows the user to specify the name to be used when writing out the comparison output files. The argument is optional and

```
"oname=<output file name>" is defined as follows:
"output file name" - Name given to the output files created for the comparison.
```

This argument is valid when the environment variable OPENSS\_CREATE\_CSV is set to 1. In this example, the comparison files created when the osscompare command is executed will be named smg\_hwc\_cmp.csv and/or smg\_hwc\_cmp.txt.

```
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" oname=mar2015_pcsamp_cmp
```

This example will generate comparison files named using the specified oname specification.

```
8 -rw-rw-r--. 1 jeg jeg 4475 Mar 11 15:53 mar2015_pcsamp_cmp.compare.csv
8 -rw-rw-r--. 1 jeg jeg 4841 Mar 11 15:53 mar2015_pcsamp_cmp.compare.txt
```

#### 11.1.4 osscompare view type or granularity argument.

osscompare allows an optional view type argument. It represents the granularity of the view. O|SS allows for viewing performance data at three levels: linked object level, function level, and at the statement level. osscompare will produce output at one of those levels based on the view type argument where:

```
"viewtype=<functions | statements | linkedobjects >" is defined as follows:
"functions" - View type granularity is per function
"statements" - View type granularity is per statement
"linkedobjects" - View type granularity is per library (linked object)
```

This example will produce a side-by-side comparison for the statement level, not the default function level. So, this example will compare statement performance values in each of the two databases and produce a side-by-side comparison showing how each statement in the application differed from the two experiments.

```
osscompare "smg2000-pcsamp.openss,smg2000-pcsamp-1.openss" viewtype=statements
```

## 12 Open|SpeedShop User Interfaces

Throughout this manual we have been using the Open|SpeedShop (O|SS) GUI, we would encourage you to play around with the interface to become familiar with it. The GUI lets you peel-off and rearrange any panel. There are also context sensitive menus so you can right click on any location to access a different view or to activate additional panels.

If you prefer not to use the GUI there are three other options that all have equal functionality. First there is the command line interface that we have also seen throughout this manual, which you can launch with the `-cli` option:

```
> openss -cli
```

There is also the immediate command (batch) interface. This uses the `-batch` flag:

```
> openss -batch < openss_cmd_file
> openss -batch -f <exe> <experiment>
```

Lastly there is a python scripting API, so you can launch O|SS commands within a python script.

> python openss\_python\_script\_file.py

## 12.1 Command Line Interface Basics

The CLI offers an interactive command line interface with processing like gdb or dbx. There are several interactive commands that allow you to create experiments, provide you with process/thread control or enable you to view experiment results. You can find the full CLI documentation at [http://www.openspeedshop.org/doc/cli\\_doc/](http://www.openspeedshop.org/doc/cli_doc/) but here we will briefly cover some important points. Here is a quick overview of some commands (those marked with a \* are only available for the online version):

<b>Experiment Creation</b> <ul style="list-style-type: none"> <li>• expcreate</li> <li>• expattach*</li> </ul>	<b>Result Presentation</b> <ul style="list-style-type: none"> <li>• expview</li> <li>• opengui</li> </ul>
<b>Experiment Control</b> <ul style="list-style-type: none"> <li>• expgo</li> <li>• expwait*</li> <li>• expdisable*</li> <li>• expenable*</li> </ul>	<b>Misc. Commands</b> <ul style="list-style-type: none"> <li>• help</li> <li>• list</li> <li>• log</li> <li>• record</li> <li>• playback</li> <li>• history</li> <li>• quit</li> </ul>
<b>Experiment Storage</b> <ul style="list-style-type: none"> <li>• expsave</li> <li>• expstore</li> </ul>	

The following is a simple example to create, run and view data from an experiment using the CLI.

> openss -cli	Open the CLI.
openss>> expcreate -f "mutatee 2000" pcsamp	Create an experiment using pcsamp with this application.
openss>> expgo	Run the experiment and create the database
openss>> expview	Display the default view of the performance data.

You can also get alternative views of the performance data within the CLI. The following is a list of some options to change the way the information is displayed.

help or help commands	Display CLI help text <sup>[1]</sup>
expview	Show the default view <sup>[1]</sup> for experiment
expview -v statements	Show time-consuming statements <sup>[1]</sup>
expview -v loops	Show time-consuming loops
expview -v linkedobjects	Show time spent in libraries <sup>[1]</sup>
expview -v calltrees,fullstack	See all unique call paths <sup>[1]</sup> in the application.
expview -m loadbalance	See load balance across all the ranks/threads/processes in the experiment.
expview -r <rank_num>	See data for specific rank(s)

expcompare -r 1 -r 2 -m time	Compare rank 1 to rank 2 for metric equal to "time". Other metrics are allowed. This is a usage example.
list -v metrics	See the list of optional performance data metrics.
list -v src	See the list of source files associated with experiment.
list -v obj	See the list of object files associated with experiment.
list -v ranks	See the list of ranks associated with experiment.
list -v hosts	See machine host names associated with experiment.
expview -m <metric>	See performance data for the metric specified.
expview -v calltrees,fullstack <experiment type> <number>	See <number> of call paths from the list of expensive call paths.
expview -v calltrees,fullstack usertime2	Shows the top two call paths in execution time.
expview <experiment-name><number>	Shows <number> of the functions from the list of the top time-consuming functions.
expview pcsamp2	Shows the two functions taking the most time.
expview -v statements <experiment-name><number>	Show <number> of the statements from the list of the top time-consuming statements
expview -Fcsv	Show the view in comma separated list format (csv)

Remember if you want the GUI at any time just issue the command **opengui** in the CLI.

### 12.1.2 CLI Metric Expressions and Derived Types

O|SS has the capability to create derived metric from the gathered metrics by using the metric expression math functionality in the command line interface (CLI). One can access the overview from the CLI by typing this help CLI command.

```

openss>>help metric_expression

*****
<metric_expression> ::= <string> ( [<constant> ||<metric_expression> ] [,
[<constant> ||<metric_expression> ] ]*)

A user defined expression that uses metrics to compute a special value for display in a report.

User defined expression can be added to an<expMetric_list>.
A functional notation is used to build the desired expression and the following, simple, arithmetic operations are
available:
Function # arguments returns
-----
Uminus() 1 unary minus of the argument
Abs() 1 Absolute value of the argument
Add() 2 summation of the arguments
Sub() 2 difference of the arguments
Mult() 2 product of the arguments
Div() 2 first argument divided by second
Mod() 2 remainder of divide operation
Min() 2 minimum of the arguments
Max() 2 maximum of the arguments
A_Add() 1 sum of all the data samples specified for the view
A_Mult() 1 product of all the data samples specified for the view
A_Min() 1 minimum of all the data samples specified for the view
A_Max() 1 maximum of all the data samples specified for the view

```

```

Sqrt()      1 square root of the argument
Stdev()     3 standard deviation calculation
Percent()   2 percent the first argument is of the second
Condexp()   3 "C" expression: "(first argument) ? second argument: third argument"
Header()    2 use the first argument as a column header for the display of the second

```

Note:

Integer and floating constants are supported as arguments as are the metric keywords associated with the experiment view.

Arguments to these functions can be <metric\_expressions>, with the exception of the first argument of 'Header'.

The first argument of 'Header' must be a character string that is preceded with and followed by '\ '.

When the '-v summary' option is used, it is not generally possible to produce a meaningful column summary. A summary is produced for Add(), Max(), Min(), Percent(), A\_Add(), A\_Max and A\_Min().

Examples:

```

expview hwc -m count,Header("\percent of counts",Percent(count,A_Add(count)) -v summary
expview mpi -v butterfly -f MPI_Alltoallv -m time,Header("average time/count",Div(Mult(time,1000),counts))
expview -m papi_l2_tca,papi_l2_tcm,Header("\percent of l2_tcm/l2_tca", Percent(papi_l2_tcm,papi_l2_tca))

```

To examine an example, we take the default view, expview command and add the capability to add the percentage that each function contributes to the total.

Add the header by using the "Header" phrase to create a header for the new data column that is being added. The "Percent" phrase to create the arithmetic expression that divides the PAPI\_L1\_DCM counts (count) for each function by the total number of PAPI\_L1\_DCM counts in the application(A\_Add(count)).

```

openss>>expview -m count,Header("\percent of counts",Percent(count,A_Add(count)))

```

```

Exclusive      percent  Function (defining location)
PAPI_L1_DCM of counts
Counts
342000000  52.333588  hypre_SMGResidual (smg2000: smg_residual.c,152)
207500000  31.752104  hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
 20500000  3.136955  hypre_SemiInterp (smg2000: semi_interp.c,126)
 15000000  2.295333  hypre_SemiRestrict (smg2000: semi_restrict.c,125)
  8500000  1.300689  pack_predefined_data (libmpi.so.0.0.3)
  7000000  1.071155  unpack_predefined_data (libmpi.so.0.0.3)

```

Another example, this one based in the hwcsamp experiment view, shows the ratio between total cache accesses and total cache misses. We create a header that is defined by the Header clause.

```

openss>>expview -m papi_l2_tca,papi_l2_tcm,Header("\percent of l2_tcm/l2_tca", Percent(papi_l2_tcm,papi_l2_tca))

```

```

papi_l2_tca papi_l2_tcm  percent of      Function (defining location)
                l2_tcm/l2_tca
289946516 109226440  37.671237  hypre_SMGResidual (smg2000: smg_residual.c,152)
203463495  74795126  36.760956  hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
 34442810 12746112  37.006597  mca_btl_vader_check_fboxes (libmpi.so.1.4.0: btl_vader_fbox.h,108)

```

### 12.1.3 CLI Automatically Generated Derived Metrics and CLI Derived Metric Names

The CLI view code has logic to match up existing PAPI hardware counters with other PAPI hardware counters, if the user specified a combination of counters that OJSS has been coded to recognize. These combinations of counters have been requested by users as combinations that would be interesting to have pre-computed and output in the CLI views.

The list of automatically generated and displayed derived metric values are discussed in the following paragraphs. A short list of the metric name that can be used in the CLI view and the hardware counters needed is displayed below:

Computational Intensity	-m intensity	PAPI_TOT_INS/PAPI_TOT_CYC
	-m l1dcmis	PAPI_L1_DCM/PAPI_L1_TCA
	-m l2tcmis	PAPI_L2_TCM/PAPI_L2_TCA
	-m l3tcmis	PAPI_L3_TCM/PAPI_L3_TCA

#### 12.1.3.1 Computational Intensity

For this derived metric, a ratio is created based on the number of total instructions (PAPI\_TOT\_INS) divided by the PAPI\_TOT\_CYC hardware counter value. This ratio gives an idea of the instruction execution computational intensity. TBD.

#### 12.1.3.2 Level 1 Data Cache Miss Ratio

For this derived metric, a ratio is created based on the number of total level 1 cache accesses with the level 1 data cache misses. This ratio gives... TBD.

#### 12.1.3.3 Level 2 Data Cache Miss Ratio

For this derived metric, a ratio is created based on the number of total level 2 cache accesses with the level 2 data cache misses. This ratio gives... TBD.

#### 12.1.3.4 Level 3 Data Cache Miss Ratio

For this derived metric, a ratio is created based on the number of total level 3 cache accesses with the level 3 data cache misses. This ratio gives... TBD.

## 12.2 CLI Batch Scripting

If you have a known set of command, you want to issue you can create a plain text file with CLI commands. For example, we create a batch file that will create, run then view the pcsamp experiment run on the application fred.

```
# Create batch file commands
> echo expcreate -f fred pcsamp >> input.script
> echo expgo >> input.script
> echo expview pcsamp10 >> input.script
```

Now to run the batch file input.script we use the `-batch` option to openss.

```
> openss -batch < input.script
```

Note that currently, in this context, this interface is only supported via the online version of O|SS, so it must have been build with the `OPENSS_INSTRUMENTOR=mrnet` options.

### 12.3 Python Scripting

The O|SS python API allows users to execute the same interactive/batch commands directly through python. Users can intersperse the normal python code with commands to O|SS. Currently this interface is only supported via the online version of O|SS.

### 12.4 MPI\_Pcontrol Support

O|SS also supports the `MPI_Pcontrol` function. This feature allows the user to gather performance data only for sections of their code bounded by the `MPI_Pcontrol` calls.

The `MPI_Pcontrol` must be added to the source code of the application.

`MPI_Pcontrol(1)` enables the gathering of performance data and `MPI_Pcontrol(0)` disables the gathering.

You must also set the O|SS environment variable `OPENSS_ENABLE_MPI_PCONTROL` to 1 in order to activate the `MPI_Pcontrol` call recognition, otherwise it will be ignored.

Optionally you can set the `OPENSS_START_ENABLED` environment variable to 1 to have performance data gathered until a `MPI_Pcontrol(0)` call is encountered.

If `OPENSS_START_ENABLED` is no set no performance data will be gathered until a `MPI_Pcontrol(1)` call is encountered.

Note that for `OPENSS_START_ENABLED` to have any effect, `OPENSS_ENABLE_MPI_PCONTROL` must be set.



"I"	Information	This option shows the metadata for the experiment. Information such as the experiment type, processes, ranks, threads, hosts, and other experiment specific information is displayed.
"U"	Update	This option updates the information in the Stats Panel display. This can be used to display any new data that may have come from the nodes on which the application is running.
"CL"	Clear auxiliary information	Clear auxiliary information. If the user has chosen a time segment of the performance data or a specific function to view the data for. This option clears the settings for that and allows the next view selection to show data for the entire program again.
"D"	Default View	The default view icon shows the performance results based on the view choice granularity selection.
"S, down arrow"	Statements per Function	Show the performance results related back to the source statements in the application for the selected function. Highlight a function in the Stats Panel and click on this icon.
"C, plus sign"	Call paths w/o coalescing	Show all the calling paths in this application. Duplicate paths will not be coalesced. All of the calling paths will be shown in their entirety.
"C, plus sign, down arrow"	Call paths w/o coalescing per Function	Show all the calling paths in this application for the selected function only. Highlight a function in the Stats Panel and click on this icon. Duplicate paths will not be coalesced. All of the calling paths will be shown in their entirety.
"HC"	Hot Call Path	Show the call path in the application that took the most time. This is a short cut to find the "hot" call path.
"B"	Butterfly view	Show the butterfly view, which displays the callers and callees of the selected function. Highlight a function in the Stats Panel and click on this icon. Then repeat to "drill" down into the callers and/or callees.
"TS"	Time Segment	Show a portion of the performance data results based on the time segment selected.
"OV"	Optional View	Use this dialog to select which performance metrics to be shown in the new performance data report.
"SA"	Source Annotation	Choose which metric to use in the source panel to annotate the source. Defaults are different for each experiment, but mostly: time.
"LB"	Load Balance	Show the load balance view, which displays the min, max, and average performance values for the application. Only available on threaded or multiple process applications.
"CA"	Cluster Analysis	Show the comparative analysis view, which displays the output of a cluster analysis algorithm run against the threaded or multiple process performance analysis results for the user application. The goal of this view is to find outlying threads or processes and report the groups of like performing threads, processes, or ranks.
"CC"	Custom Compare	Raise the custom comparison panel, which provides mechanisms allowing the user to create custom views of the performance analysis results. This allows the user to supplement the provided OJSS views.

### 12.5.1.2 View/Display Choice Selection

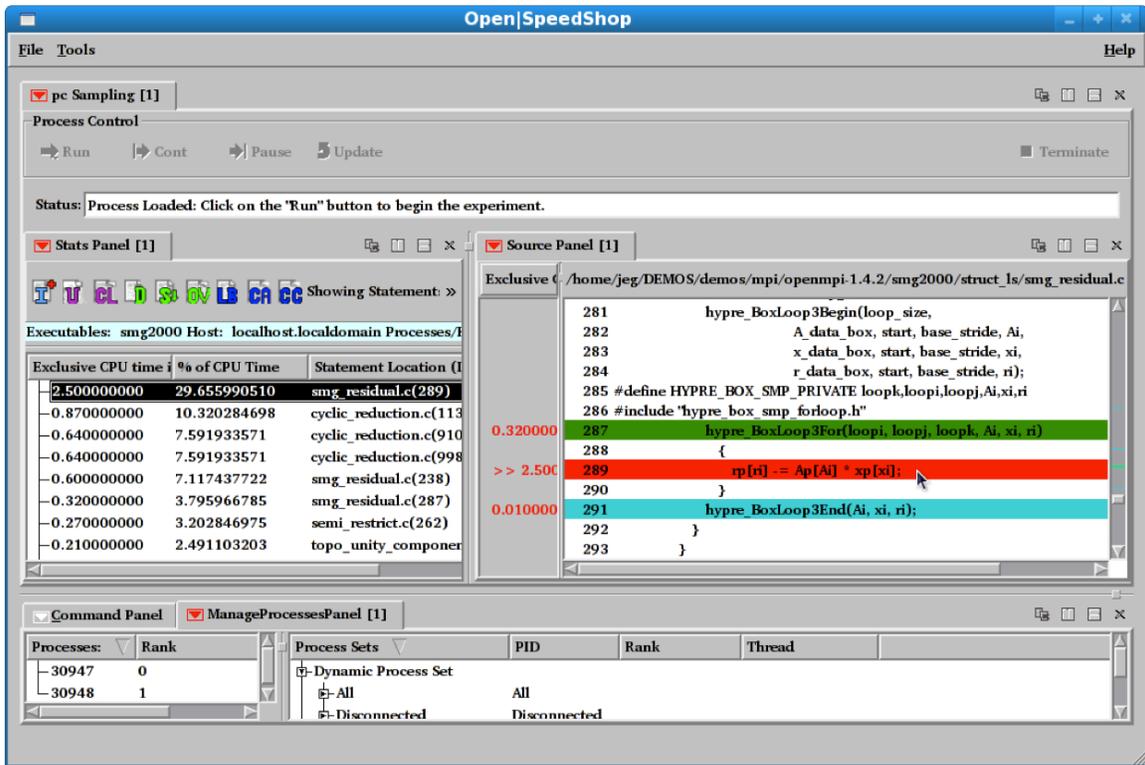
The View/Display Choice set of buttons allows users to choose what granularity to use for a particular display. The normal usage scenario, is to choose a view choice

granularity and then select a view by choosing one of the icons described in the table above. The choices, as shown in the image below, are to see the performance data displayed:

- Per Function – Display the performance information relative to each function in the program that had performance data gathered during the experiment that was run.
- Per Statement – Display the performance information relative to each statement in the program that had performance data gathered during the experiment that was run
- Per Linked Object – Display the performance information relative to each library or linked object in the program that had performance data gathered during the experiment that was run.
- Per Loop – Display the performance information relative to each loop in the program that had performance data gathered during the experiment that was run. Note that the loop performance information is only shown for loops that actually were executed. There may be loops in the application that will not show up in the display because they did not execute or had minimal time attributed to them.

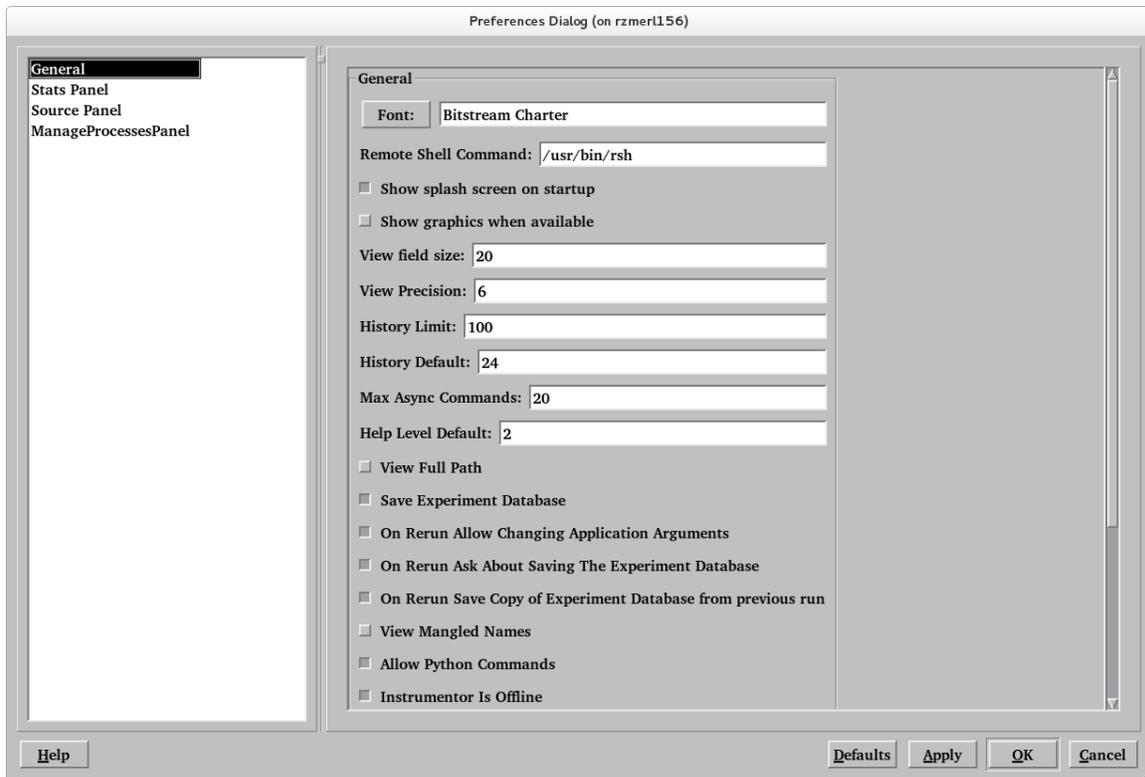


The image below illustrates that double clicking on a line of statistical information in the Stats Panel will focus the source panel at the line of source representing the performance information and annotates the source with that information. Note the hot to cold color highlighting of the source. The higher the performance values are the hotter the color. Red is the hottest color, so source highlighted in red is taking the most time in the program being profiled.

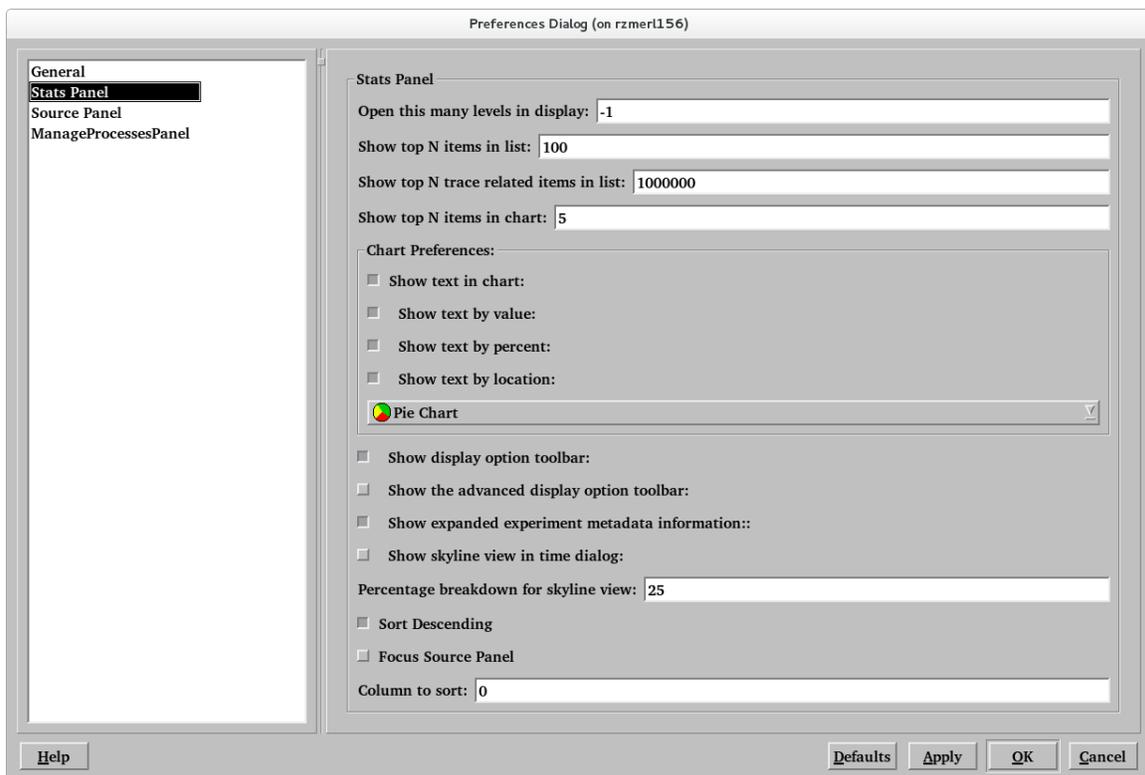


## 12.5.2 Preferences - How to change preferences

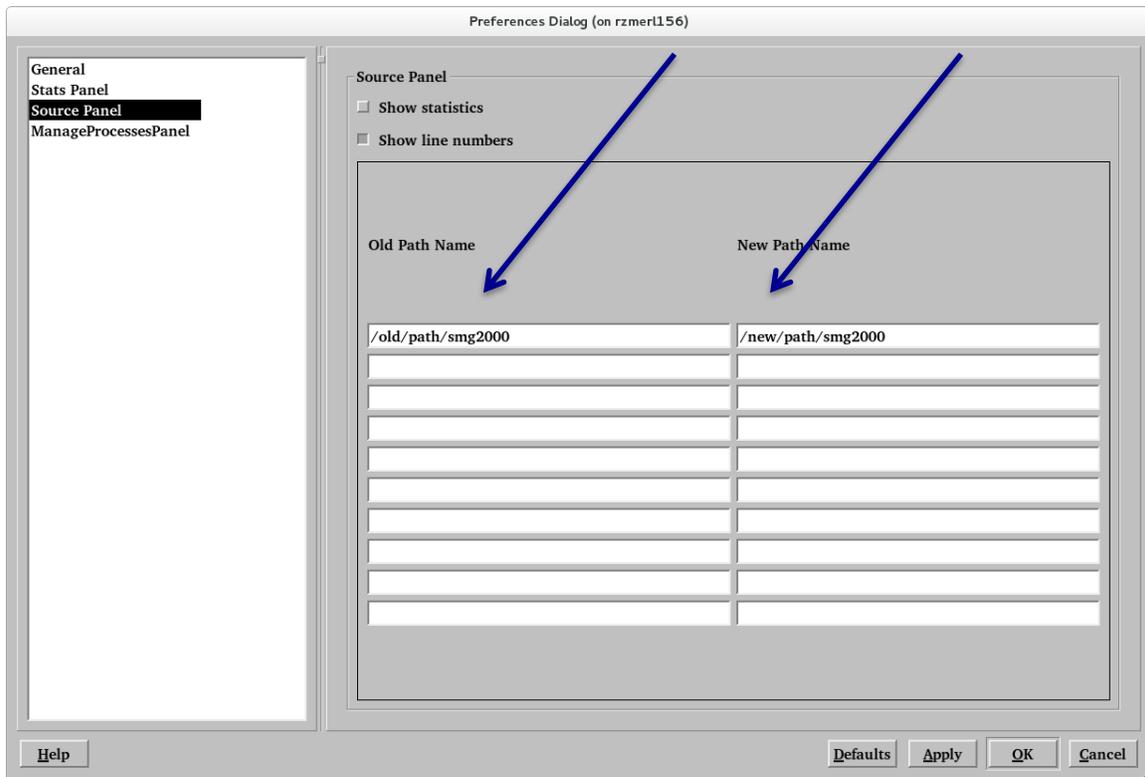
These preference panel views are included to outline the sequence for changing the GUI and CLI options for generating and viewing performance information with O|SS. The first view is the main (General) preference panel which allows the font, view field sizes, data precision, path, number of lines in the view, and many other general options.



The Stats Panel preference panel lets you change preferences related to viewing the performance information in the GUI Stats Panel.

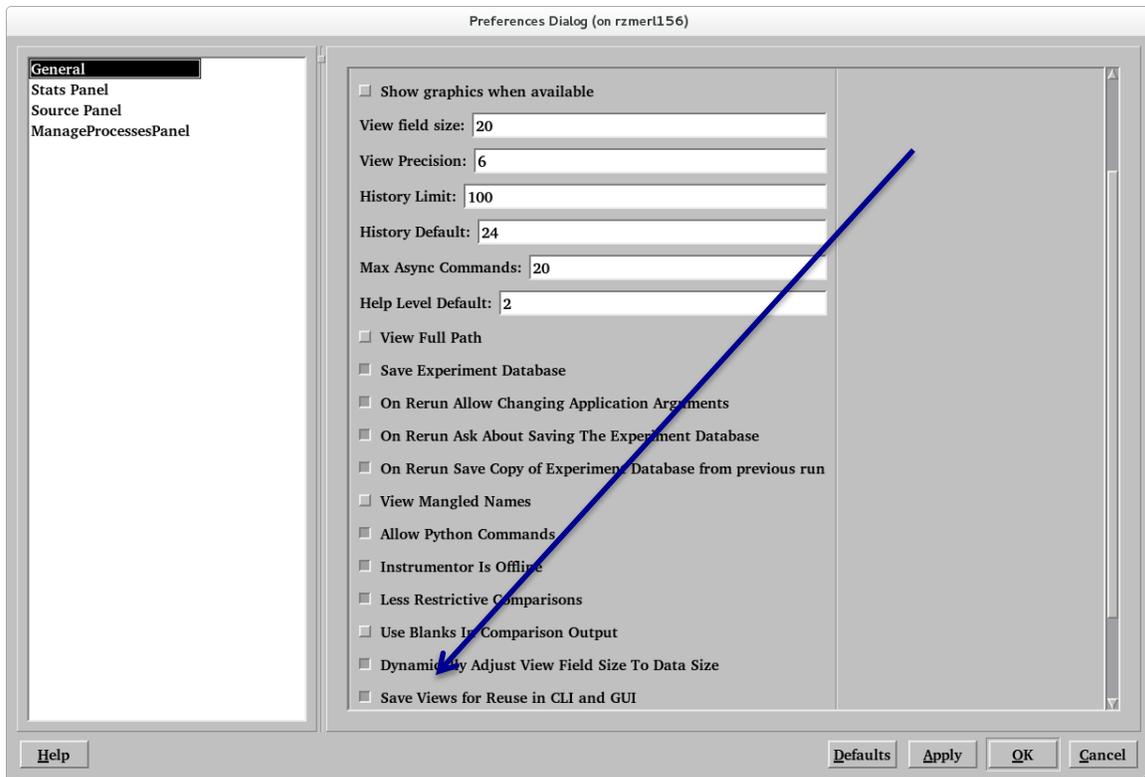


The Source Panel preference panel lets you remap paths to source files that are in a different location on the viewing platform. Use this when you can't see the source files on the viewing machine because the executable was built on a different machine. Put the old path to the source into the Old Path Name text box area and the new path for the source on the viewing machine into the New Path Name text box area.



### 12.5.2.1 Disabling or enabling the preference for Save/Reuse views in CLI.

Here we show the General preferences window scrolled down to the area of the view that shows more preference options. If you do not want the new save/reuse view active, then you can disable that function by clicking on the “Save Views for Reuse in CLI and GUI” (pointed to by the blue line below). By clicking on that preference line you can disable or enable the feature. The same procedure works for the other preferences as well.



## 13 Special System Support (Static Executables)

### 13.1 Cray and Blue Gene

When shared library support is limited the normal manner of running experiments in O|SS doesn't work. You must link the collectors into the static executable. Currently O|SS has static support on Cray and the Blue Gene P/Q platforms. You must relink the application with the `osslink` command to add support for the collectors.

The `osslink` command is a script that will help with linking. Calls to it are usually embedded inside an application's makefile. The user generally needs to fine the target that creates the actual static executable and create a collector target that links in the selected collector. The following is an example for re-linking the `smg2000` application.

```
smg2000: smg2000.o
    @echo "Linking" "$@" "... "
    ${CC} -o smg2000 smg2000.o ${LFLAGS}

smg2000-pcsamp: smg2000.o
    @echo "Linking" "$@" "... "
    osslink -v -c pcsamp ${CC} -o smg2000-pcsamp smg2000.o ${LFLAGS}

smg2000-usertime: smg2000.o
    @echo "Linking" "$@" "... "
    osslink -v -c usertime ${CC} -o smg2000-usertime smg2000.o ${LFLAGS}

smg2000-hwcsamp: smg2000.o
    @echo "Linking" "$@" "... "
    osslink -v -c hwcsamp ${CC} -o smg2000-hwcsamp smg2000.o ${LFLAGS}

smg2000-io: smg2000.o
    @echo "Linking" "$@" "... "
    osslink -v -c io ${CC} -o smg2000-io smg2000.o ${LFLAGS}

smg2000-iot: smg2000.o
    @echo "Linking" "$@" "... "
    osslink -v -c iot ${CC} -o smg2000-iot smg2000.o ${LFLAGS}

smg2000-mpi: smg2000.o
    @echo "Linking" "$@" "... "
    osslink -v -c mpi ${CC} -o smg2000-mpi smg2000.o ${LFLAGS}
```

Running the re-linked executable will cause the application to write the raw data files to the location specified by the environment variable `OPENSS_RAWDATA_DIR`. Normally, in the cluster environment where shared/dynamic executables are being run, the conversion from raw data to an O|SS database is done under the hood. However, in this case you must use the `ossutil` command to create the database file manually. Of course you can add the `ossutil` command to a batch script to eliminate

the step of manually issuing that command. Once you have the O|SS database files create you can view them normally with the GUI or CLI.

Below is an example of a job script that will execute these steps for you.

```
#PBS -q debug
#PBS -N smg2000-pcsamp
...
# must have a clean raw data directory each run
rm -rf /home/USER/smg2000/test/raw
mkdir /home/USER/smg2000/test/raw

setenv OPENSS_RAWDATA_DIR /home/USER/smg2000/test/raw
setenv OPENSS_DB_DIR /home/USER/smg2000/test/

cd /home/jgalaro/smg2000/test

# needs -b to have the original executable path available and match where
# the application was run when doing ossutil
aprun -b -n 16 /home/USER/smg2000/test/smg2000-pcsamp

# creates a X.0.openss database file, please
# load the module pointing to openspeedshop before accessing ossutil
ossutil /home/jgalaro/smg2000/test/raw
```

O|SS needs the executable path that is used to process symbols after the run is complete, to match where the executable was run. The executable path must match the path that is in the raw data that is written to the directory represented by `OPENSS_RAWDATA_DIR`. If the aprun “-b” option is not used, then the executable is run in a temporary system directory and the raw data reflects that directory path for the executable instead of the path where the executable is located when the job is initiated. This will cause ossutil to be unable to resolve the symbols.

There have been recent changes to the shared library support in O|SS. Dynamic shared library support is now available in newer Cray and Blue Gene operating systems. There is support for both shared and static binaries on the Cray and on the Blue Gene Q platforms.

### 13.1.1 osslink Command Information

The osslink command links the O|SS collectors and runtime libraries into the static executable and manages the setting the appropriate libraries based on the collector value that is one of the inputs to osslink. The help output for osslink follows.

```
osslink --help

Usage: /opt/osscbtf_cmake_only_july10/bin/osslink -c collector [options] compiler file ...

-h, --help
```

-c, --collector <collector name>

Where collector is the name of the OpenSpeedShop collector to link into the application. See the openss man page for a description of the available experiments provided by OpenSpeedShop. This is a mandatory option.

-i | --mpitype

For MPI experiments, set the OPENS\_S\_M\_P\_I\_I\_M\_P\_L\_E\_M\_E\_N\_T\_A\_T\_I\_O\_N value to the MPI implementation specified. Valid options are:

- mpich
- mpich2
- mvapich
- mvapich2
- openmpi
- mpt
- lam
- lampi

-v, --verbose

### 13.1.2 Cray Specific Static aprun Information

Note, in the above execution of the statically linked executable that we need to add the `-b` option to the aprun call. The option is needed because O|SS stores information about the executable location when it is running. Without the `-b` option the executable is run in a temporary location that is not available when the raw data information is being converted into the O|SS database file.

### 13.1.3 Changing parameters to the experiments

Note, when execution of the statically linked executable with the O|SS collectors linked in, the workflow is different. Since the more flexible convenience scripts can't be used, in order to change the arguments to the experiments, one must set environment variables.

Examples of the environment variables that can be changed are as follows:

Environment Variable	Represents	Experiment Type
OPENS_S_P_C_S_A_M_P_R_A_T_E	Sampling Rate.	pcsamp
OPENS_S_U_S_E_R_T_I_M_E_R_A_T_E	Sampling Rate.	usertime
OPENS_S_H_W_C_E_V_E_N_T	PAPI or Native Event Name.	hwc
OPENS_S_H_W_C_T_H_R_E_S_H_O_L_D	How many events occurrences before sample taken.	hwc
OPENS_S_H_W_C_S_A_M_P_E_V_E_N_T_S	List of PAPI or Native Event Names.	hwcsamp
OPENS_S_H_W_C_S_A_M_P_R_A_T_E	Sampling Rate.	hwcsamp

OPENSS_HWCTIME_EVENT	PAPI or Native Event Name.	hwctime
OPENSS_HWCTIME_THRESHOLD	How many events occurrences before sample taken.	hwctime
OPENSS_IO_TRACED	List of I/O functions to collect data for.	io
OPENSS_IOT_TRACED	List of I/O functions to collect data for.	iot
OPENSS_MPI_TRACED	List of MPI functions to collect data for.	mpi
OPENSS_MPIT_TRACED	List of MPI functions to collect data for.	mpit

## 14 Setup and Build for Open|SpeedShop

Open|SpeedShop (O|SS) is setup to work with a variety of processor types including Intel, AMD, Intel Phi, PPC, and ARM processor architectures. It has been tested on many Linux Distributions include SLES, SUSE, RHEL, Fedora Core, CentOS, Debian, Ubuntu and many others. It has been installed on the IBM Blue Gene and the Cray systems. The O|SS website contains information on special builds and usage instructions.

The source code for O|SS is available for download at the O|SS project home on Sourceforge:

<http://sourceforge.net/projects/openss>

Or CVS access is available at:

[http://sourceforge.net/scm/?type=cvs&group\\_id=176777](http://sourceforge.net/scm/?type=cvs&group_id=176777)

Packages and additional information can be found on the O|SS website:

<http://www.openspeedshop.org/>

### 14.1 Open|SpeedShop Cluster Install

Open|SpeedShop (O|SS) comes with a set of bash install scripts that will build O|SS and any components it needs from source tarballs. First it will check to see if the correct supporting software is installed on your system, if the needed software isn't installed it will ask to build it for you. The only thing you need to do is provide a few arguments for the install script. For a normal setup you would just specify the directory to install in, what build task you want to do, and the location of your MPI and QT installs. For example:

```
./install-tool --build-krell-root --krell-root-prefix /opt/krellroot --with-openmpi /opt/openmpi-1.8.2 --with-mvapich /opt/mvapich-1.7
```

```
./install-tool --build-offline --openss-prefix /opt/myoss --krell-root-prefix /opt/krellroot --with-openmpi /opt/openmpi-1.8.2 --with-mvapich /opt/mvapich-1.7
```

After the install has successfully completed there are a few important environment variables you need to set. Set a variable for the install location, so you can reuse it. Then set the OPENSS\_PLUGIN\_PATH for the directory where the plugins are stored. If you installed with more than one MPI version, you must specify which to use with OPENSS\_MPI\_IMPLEMENTATION. Lastly, add the O|SS and Krell externals (root) bin directory to your PATH and lib64 directories to your LD\_LIBRARY\_PATH. Examples of the necessary environment variables that need to be set are as follows:

```
export oss_install_dir=/opt/myoss
export root_install-dir=/opt/krellroot
export OPENSS_MPI_IMPLEMENTATION=openmpi
export OPENSS_PLUGIN_PATH=$oss_install_dir/lib64/openspeedshop
```

```

export LD_LIBRARY_PATH=$root_install_dir/lib64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=$oss_install_dir/lib64:$LD_LIBRARY_PATH
export DYNINSTAPI_RT_LIB=$root_install_dir/lib64/libdyninstAPI_RT.so
export PATH=$root_install_dir/bin:$PATH
export PATH=$oss_install_dir/bin:$PATH

```

## 14.2 Open | SpeedShop Blue Gene Platform Install

Please reference the O|SS 2.2 Build and Install Guide.

## 14.3 Open | SpeedShop Cray Platform Install

Please reference the O|SS 2.2 Build and Install Guide.

## 14.4 Execution Runtime Environment Setup

This section gives an example of a module file, softenv file and dotkit that can be used to set-up the O|SS execution environments.

### 14.4.1 Example module file

This is an example of a module file used for a cluster installation. Use module load <filename of module file> to activate the O|SS runtime environment.

```

##%Module1.0#####
#####
## openss modulefile
##
proc ModulesHelp { } {
    global version openss
    puts stderr "\topenss - loads the OpenSpeedShop software & application environment"
    puts stderr "\n\tThis adds $oss/* to several of the"
    puts stderr "\tenvironment variables."
    puts stderr "\n\tVersion $version\n"
}
module-whatis "loads the OpenSpeedShop runtime environment"
# for Tcl script use only
set version 2.2
set oss /opt/OSS21

setenv OPENSS_PREFIX $oss
setenv OPENSS_DOC_DIR $oss/share/doc/packages/OpenSpeedShop
prepend-path PATH $oss/bin
prepend-path MANPATH $oss/share/man

set unnameexe "/bin/uname"
if { [file exists $unnameexe] } {

```

```

set machinetype [ exec /bin/uname -m ]
if { $machinetype == "x86" ||
    $machinetype == "i386" ||
    $machinetype == "i486" ||
    $machinetype == "i586" ||
    $machinetype == "i686" } {
    setenv OPENSS_PLUGIN_PATH $oss/lib/openspeedshop
    setenv DYNINSTAPI_RT_LIB $oss/lib/libdyninstAPI_RT.so
    prepend-path LD_LIBRARY_PATH $oss/lib
}
if { $machinetype == "x86_64" } {
    setenv OPENSS_PLUGIN_PATH $oss/lib64/openspeedshop
    setenv DYNINSTAPI_RT_LIB $oss/lib64/libdyninstAPI_RT.so
    prepend-path LD_LIBRARY_PATH $oss/lib64
}
if { $machinetype == "ia64" } {
    setenv OPENSS_PLUGIN_PATH $oss/lib/openspeedshop
    setenv DYNINSTAPI_RT_LIB $oss/lib/libdyninstAPI_RT.so
    prepend-path LD_LIBRARY_PATH $oss/lib
}
}

```

#### 14.4.2 Example softenv file

This is an example of a softenv file used for a Blue Gene/Q installation. Use the “resoft <filename of softenv file>” command to activate the OJSS runtime environment.

```

# The OpenSpeedShop .soft file.
# Remember to type "resoft" after working on this file.

OSS = /home/projects/oss/oss
TARCH = bgq

# Set up OSS environment variables

# Find the executable portions of OpenSpeedShop (order is important here)
PATH += $OSS/$TARCH/bin
PATH += $OSS/bin

# Find the libraries for OpenSpeedShop (order is important here)
LD_LIBRARY_PATH += $OSS/$TARCH/lib64
LD_LIBRARY_PATH += $OSS/lib64

# Find the runtime collectors
OPENSS_PLUGIN_PATH = $OSS/$TARCH/lib64/openspeedshop

# Tell the tool what the application MPI implementation is
# Needed if supporting multiple implementations and running the "mpi", "mpit" or "mpip" experiments
OPENSS_MPI_IMPLEMENTATION = mpich2

# Paths to documentation and man pages
OPENSS_DOC_DIR = $OSS/share/doc/packages/OpenSpeedShop
MANPATH = $OSS/share/man

```

```
# Use the basic environment.  
@default
```

### 14.4.3 Example dotkit file

This is an example of a dotkit file used for a 64-bit cluster platform installation and is not generalized to support different platforms other than the 64-bit cluster it was written for. Use the “use <filename of dotkit file>” command to activate the OSS runtime environment. Note: do not include the “.dk” portion of the filename when using the “use” command.

```
#c performance/profile  
#d Open|SpeedShop (Version 2.2)  
dk_setenv OPENSS_PREFIX /usr/global/tools/openspeedshop/oss-dev/OSS21  
dk_setenv OPENSS_PLUGIN_PATH $OPENSS_PREFIX/lib64/openspeedshop  
dk_setenv OPENSS_DOC $OPENSS_PREFIX/share/doc/packages/OpenSpeedShop/  
dk_alter PATH $OPENSS_PREFIX/bin  
dk_alter LD_LIBRARY_PATH $OPENSS_PREFIX/lib64  
  
dk_setenv DYNINSTAPI_RT_LIB $OPENSS_PREFIX/lib64/libdyninstAPI_RT.so  
dk_setenv XPLAT_RSH rsh  
dk_setenv OPENSS_MPI_IMPLEMENTATION mvapich  
dk_test `dk_cev OPENSS_RAWDATA_DIR` -eq 0 && dk_setenv OPENSS_RAWDATA_DIR  
/p/lscratchb/${USER}
```

## 15 Additional Information and Documentation Sources

### 15.1 Final Experiment Overview

In the table below we match up a few general questions you may ask yourself with the experiments you may want to run in order to find the answer.

Where does my code spend most of its time?
<ul style="list-style-type: none"><li>• Flat profiles (pcsamp)</li><li>• Getting inclusive/exclusive timings with call paths (usertime)</li><li>• Identifying hot call paths (usertime + HP analysis)</li></ul>
How do I analyze cache performance?
<ul style="list-style-type: none"><li>• Measure memory performance using hardware counters (hwc)</li><li>• Compare to flat profiles (custom comparison)</li><li>• Compare multiple hardware counters (N x hwc, hwcsamp)</li></ul>
How to identify I/O problems?
<ul style="list-style-type: none"><li>• Study time spent in I/O routines (io, iot and lightweight iop)</li><li>• Compare runs under different scenarios (custom comparisons)</li></ul>
How to identify memory problems?
<ul style="list-style-type: none"><li>• Study time spent in memory allocation/de-allocation routines (mem)</li><li>• Look for load imbalance (LB view) and outliers (CA view)</li></ul>
How do I find parallel inefficiencies in OpenMP and/or threaded applications?
<ul style="list-style-type: none"><li>• Study time spent in POSIX thread routines (pthreads)</li><li>• Look for load imbalance (LB view) and outliers (CA view)</li></ul>

How do I find parallel inefficiencies in MPI applications?
<ul style="list-style-type: none"> <li>• Study time spent in MPI routines (mpi, mpit, and lightweight mpip)</li> <li>• Look for load imbalance (LB view) and outliers (CA view)</li> </ul>
How do I find parallel inefficiencies in NVIDIA CUDA applications?
<ul style="list-style-type: none"> <li>• Study time spent in CUDA routines and the CUDA event execution trace. (cuda)</li> </ul>

## 15.2 Additional Documentation

The python scripting API documentation can be found at [http://www.openspeedshop.org/docs/pyscripting\\_doc](http://www.openspeedshop.org/docs/pyscripting_doc) or in the `.../share/doc/packages/openspeedshop/pyscripting_doc` folder in the install directory.

There are also man pages for openss and every convenience script. There's also a quick start guide that you can download from <http://www.openspeedshop.org>

There is also an O|SS Forum type email alias, where you can ask questions and read previous posts at [oss-questions@openspeedshop.org](mailto:oss-questions@openspeedshop.org). Use this URL to sign up: <https://groups.google.com/a/krellinst.org/forum/?hl=en-!forum/oss-questions>

There is also an email list that you can send your questions to without joining the group. The email alias is [oss-contact@openspeedshop.org](mailto:oss-contact@openspeedshop.org).

## 16 Convenience Script Basic Usage Reference Information

This section provides a quick overview of the convenience scripts that can be used to either compare experiment data to other experiment data or to gather performance information for each of the various performance metric types that O|SS supports.

### 16.1 Suggested Workflow

We recommend an O|SS workflow consisting of two phases. First, gathering the performance data using the convenience scripts. Then using the GUI or CLI to view the data.

### 16.2 Convenience Scripts

Users are encouraged to use the convenience scripts (for dynamically linked applications) that hide some of the underlying options for running experiments. The full command syntax can be found in the User's Guide. The script names correspond to the experiment types and are: **osspsamp**, **ossusertime**, **osshwc**, **osshwcsamp**, **osshwctime**, **ossio**, **ossiop**, **ossmipi**, **ossmpt**, **ossmem**, **ossmem**, **ossmptp**, **osspthreads**, **osmpip**, and **ossCUDA** plus an **osscompare** script. Note: If using the offline mode of operation, make sure to set **OPENSS\_RAWDATA\_DIR** (See **KEY ENVIRONMENT VARIABLES** section for info).

When running Open|SpeedShop, use the same syntax that is used to run the application/executable outside of O|SS, but enclosed in quotes; e.g.,

Using MPI drivers like mpirun: **osspsamp "mpirun -np 512 ./smg2000 -n 5 5 5"**

Using SLURM/srun: **osspsamp "srun -N 64 -n 512 ./smg2000 -n 5 5 5"**

Redirection to/from files inside quotes can be problematic, see convenience script "man" pages for more info.

### 16.3 Report and Database Creation

Running the pcsamp experiment on the sequential program named mexe: **osspsamp mexe** results in a default report and the creation of a SQLite database file **mexe-pcsamp.openss** in the current directory; the report:

% CPU Time	CPU time	Function
48.990	11.650	f3 (mexe: m.c, 24)
33.478	7.960	f2 (mexe: m.c,15)
17.451	4.150	f1 (mexe: m.c,6)
0.084	0.020	work(mexe:m.c,33)

To access alternative views in the GUI: **openss -f mexe-pcsamp.openss** loads the database file. Then use the GUI toolbar to select desired views; or, using the CLI: **openss -cli -f mexe-pcsamp.openss** to load the database file. Then use the **expview** command options for desired views.

## 16.4 osscompare: Compare Database Files

General form:

```
osscompare "<db_file1>, <db_file2>[,<db_file>...]" [ time | percent | <other metrics>] [rows=nn]
[viewtype=functions| statements | linkedobjects ] > [ oname = <csv filename> ]
```

Where:

"<db\_file>" represents an O|SS database file created by running an O|SS experiment on an application.

[ time | percent | <other metrics>] represent the metric that the comparison will use to differentiate the performance information for each experiment database.

[rows=nn] indicates how many rows of output you want to have listed.

[viewtype=functions| statements | linkedobjects] select the granularity of the view output. The comparison is either done at the function, statement, or library view level. Function level is the default granularity.

[ oname = <csv filename> ] Name the output filename when comma separated list output is requested.

Example:

```
osscompare "smg-run1.openss,smg-run2.openss" SEP percent rows=10
osscompare "smg-run1.openss,smg-run2.openss" percent rows=10
```

Please type "man osscompare" for more details.

## 16.5 osspcsamp: Program Counter Experiment

General form:

```
osspcsamp "<command> < args>" [ high | low | default | <sampling rate>]
```

Sequential job example:

```
osspcsamp "smg2000 -n 50 50 50"
```

Parallel job example:

```
osspcsamp "mpirun -np 128 smg2000 -n 50 50 50"
```

Additional arguments:

**high:** twice the default sampling rate (samples per second)

**low:** half the default sampling rate

**default:** default sampling rate is 100

<sampling rate>: integer value sampling rate

## 16.6 ossusertime: Call Path Experiment

General form:

```
ossusertime "<command> < args>" [ high | low | default | <sampling rate>]
```

Sequential job example:

```
ossusertime "smg2000 -n 50 50 50"f L IsSEP
```

Parallel job example:

```
ossusertime "mpirun -np 64 smg2000 -n 50 50 50"
```

Additional arguments:

**high:** twice the default sampling rate (samples per second)

**low:** half the default sampling rate

**default:** default sampling rate is 35

<sampling rate>: integer value sampling rate

## 16.7 osshwc, osshwctime: HWC Experiments

General form:

```
osshwc[time] "<command> <args>" [ default | <PAPI_event> | <PAPI threshold> |  
<PAPI_event><PAPI threshold> ]
```

Sequential job example:

```
osshwc[time] "smg2000 -n 50 50 50"f L IsSEP
```

Parallel job example:

```
osshwc[time] "mpirun -np 128 smg2000 -n 50 50 50"f L IsSEP
```

Additional arguments:

**default:** event (PAPI\_TOT\_CYC), threshold (10000)

<PAPI\_event>: PAPI event name

<PAPI threshold>: PAPI integer threshold

## 16.8 osshwcsamp: HWC Experiment

General form:

```
osshwcsamp "<command><args>" [ default | <PAPI_event_list> | <sampling_rate> ]
```

Sequential job example:

```
osshwcsamp "smg2000 -n 50 50 50"
```

Parallel job examples:

```
osshwcsamp "mpirun -np 128 smg2000 -n 50 50 50"
```

```
osshwcsamp "srun -N 32 -n 128 sweep3d.mpi" PAPI_L1_DCM,PAPI_L1_DCA 200
```

Additional arguments:

**default:** events(PAPI\_TOT\_CYC and PAPI\_FP\_OPS), sampling\_rate is 100

<PAPI\_event\_list>: Comma separated PAPI event list<sup>f L I</sup><sub>sSEP</sub>; <sampling\_rate>: Integer value  
sampling rate

## 16.9 ossio, ossiot: I/O Experiments

General form:

**ossio[t]** "<command> < args>" [ **default** | f\_t\_list ]

Sequential job example:

**ossio[t]** "smg2000 -n 50 50 50"

Parallel job example:

**ossio[t]** "mpirun -np 128 smg2000 -n 50 50 50"

Additional arguments:

**default:** trace all I/O functions

< f\_t\_list>: Comma-separated list of I/O functions to trace, one or more of the following: **close, creat, creat64, dup, dup2, lseek, lseek64, open, open64, pipe, pread, pread64, pwrite, pwrite64, read, readv, write, and writev**

## 16.10 ossmpi, ossmpip, ossmpit: MPI Experiments

General form:

**ossmpi[p|t]** "<mpirun><mpiargs><command><args>" [ **default** | f\_t\_list ]

Parallel job example:

**ossmpi[p|t]** "mpirun -np 128 smg2000 -n 50 50 50"

Additional arguments:

**default:** trace all MPI functions

<f\_t\_list>: Comma-separated list of MPI functions to trace, consisting of zero or more of: **MPI\_Allgather, ..., MPI\_Wait**some and/or zero or more of the MPI group categories:

MPI Category	Argument
All MPI Functions	all
Collective Communicators	collective_com
Persistent Communicators	persistent_com
Synchronous Point to Point	synchronous_p2p
Asynchronous Point to Point	asynchronous_p2p
Process Topologies <sup>[L]</sup> <sub>[SEP]</sub>	process_topologies
Groups Contexts Communicators	graphs_contexts_comms
Environment <sup>[L]</sup> <sub>[SEP]</sub>	environment <sup>[L]</sup> <sub>[SEP]</sub>
Datatypes	datatypes
MPI File I/O	fileio

## 16.11 ossmem: Memory Analysis Experiment

General form:

**ossmem** "<command> < args>" [ **default** | f\_t\_list ]

Sequential job example:

**ossmem** "smg2000 -n 50 50 50"

Parallel job example:

```
ossmem "mpirun -np 128 smg2000 -n 50 50 50"
```

Additional arguments:

**default:** trace all supported memory functions

<f\_t\_list>: Comma-separated list of exceptions to trace, consisting of one or more of: **malloc**, **free**, **memalign**, **posix\_memalign**, **calloc** and **realloc**

## 16.12 ossomptp: OpenMP Specific Profiling Experiment

General form:

```
ossomptp "<command> <args>"
```

Sequential job example:

```
ossomptp "openmp_stress < stress.input"
```

Parallel job example:

```
ossomptp "mpirun -np 128 openMP_MD"
```

## 16.13 osspthreads: POSIX Thread Analysis Experiment

General form:

```
osspthreads "<command> <args>" [default | f_t_list ]
```

Sequential job example:

```
osspthreads "smg2000 -n 50 50 50"
```

Parallel job example:

```
osspthreads "mpirun -np 128 smg2000 -n 50 50 50"
```

Additional arguments:

**default:** trace all POSIX thread functions

<f\_t\_list>: Comma-separated list of exceptions to trace, consisting of one or more of:

**pthread\_create**, **pthread\_mutex\_init**, **pthread\_mutex\_destroy**, **pthread\_mutex\_lock**, **pthread\_mutex\_trylock**, **pthread\_mutex\_unlock**, **pthread\_cond\_init**, **pthread\_cond\_destroy**, **pthread\_cond\_signal**, **pthread\_cond\_broadcast**, **pthread\_cond\_wait**, and **pthread\_cond\_timedwait**

## 16.14 osscuda: NVIDIA CUDA Tracing Experiment

General form:

```
osscuda "<command> <args>"
```

Sequential job example:

```
osscuda "eigenvalues --matrix-size=4096"
```

Parallel job example:

```
osscuda "mpirun -np 64 -npernode 1 lmp_linux -sf gpu < in.lj"
```

## 16.15 Key Environment Variables

<b><u>EXECUTION RELATED VARIABLES</u></b>	<b><u>DESCRIPTION</u></b>
<b>OPENSS_RAWDATA_DIR</b>	Used on cluster systems where a /tmp file system is unique on each node. It specifies the location of a shared file system path which is required for OJSS to save the "raw" data files on distributed systems. <b>OPENSS_RAWDATA_DIR="shared file system path"</b> Example: export <b>OPENSS_RAWDATA_DIR=/lustre4/fsys/userid</b>
<b>OPENSS_ENABLE_MPI_PCONTROL</b>	Activates the MPI_Pcontrol function recognition, otherwise MPI_Pcontrol function calls will be ignored by OJSS.
<b>OPENSS_DATABASE_ONLY</b>	When running the OJSS convenience scripts only create the database file and do NOT put out the default report. Used to reduce the size of the batch file output files if user is not interested in looking at the default report.
<b>OPENSS_RAWDATA_ONLY</b>	When running the OJSS convenience scripts only gather the performance information into the OPENSS_RAWDATA_DIR directory, but do NOT create the database file and do NOT put out the default report.
<b>OPENSS_DB_DIR</b>	Specifies the path to where OJSS will build the database file. On a file system without file locking enabled, the SQLite component cannot create the database file. This variable is used to specify a path to a file system with locking enabled for the database file creation. This usually occurs on Lustre file systems that don't have locking enabled. <b>OPENSS_DB_DIR="file system path"</b> Example: export <b>OPENSS_DB_DIR=/opt/filesys/userid</b>
<b>OPENSS_MPI_IMPLEMENTATION</b>	Specifies the MPI implementation in use by the application; only needed for the mpi, mpit, and mpip experiments. These are the currently supported MPI implementations: <b>openmpi, lampi, mpich, mpich2, mpt, lam, mvapich, mvapich2</b> . For Cray, IBM, Intel MPI implementations, use <b>mpich2</b> . <b>OPENSS_MPI_IMPLEMENTATION="MPI impl. name"</b> Example: export <b>OPENSS_MPI_IMPLEMENTATION=openmpi</b> In most cases, OJSS can auto-detect the MPI in use.
<b>OPENSS_DEFER_VIEW</b>	Allow overriding the displaying of the default view for cases where users may not want or need it to be displayed.

