



Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Contributors:

Jim Galarowicz, Bill Hachfeld, Greg Schultz: Argo Navis Technologies

Don Maghrak, Patrick Romero: Krell Institute

Martin Schulz, Matt Legendre: Lawrence Livermore National Laboratory

Jennifer Green, Dave Montoya: Los Alamos National Laboratory

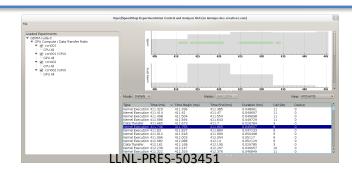
Mahesh Rajan, Doug Pase: Sandia National Laboratories

Koushik Ghosh, Engility

Dyninst group (Bart Miller: UW & Jeff Hollingsworth: UMD)

Phil Roth, Mike Brim: ORNL







Outline





- Introduction to Open | SpeedShop
- How to run basic timing experiments and what they can do?
- How to deal with parallelism (MPI and threads)?
- How to properly use hardware counters?
- Slightly more advanced targets for analysis
 - How to understand and optimize I/O activity?
 - How to evaluate memory efficiency?
 - How to analyze codes running on GPUs?
- DIY and Conclusions: DIY and Future trends
- Hands-on Exercises (after each section)
 - On site cluster available
 - We will provide exercises and test codes





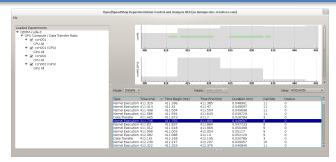
Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 1 Introduction to Open|SpeedShop







Open | SpeedShop Tool Set





Open Source Performance Analysis Tool Framework

- > Most common performance analysis steps all in one tool
- Combines tracing and sampling techniques
- Gathers and displays several types of performance information

Flexible and Easy to use

User access through:
GUI, Command Line, Python Scripting, convenience scripts

Scalable Data Collection

- > Instrumentation of *unmodified application binaries*
- > New option for *hierarchical online data aggregation*

Supports a wide range of systems

- > Extensively used and tested on a variety of *Linux clusters*
- > Cray, Blue Gene, ARM, Power 8, Intel Phi, GPU support

Classifying Open | SpeedShop





Offers both sampling and direct instrumentation

- > Sampling for overview and hardware counter experiments
 - Even and low overhead, overview information
- > Direct instrumentation for more detailed experiments
 - More in-depth information, but potentially bursty
- > All instrumentation at link-time of runtime

Multiple direct instrumentation options

- > API level instrumentation (e.g., I/O or memory)
- > Loop analysis based on binary instrumentation techniques
- Programming model specific instrumentation (e.g., MPI or OpenMP)

Aggregation

- > By default: aggregate profile data over time
 - Example: intervals, functions, ...
 - Full traces possible for some experiments (e.g., MPI), but costly
- > For parallel experiments: by default aggregation over threads, processes, ...
 - However, users can query per process/thread data

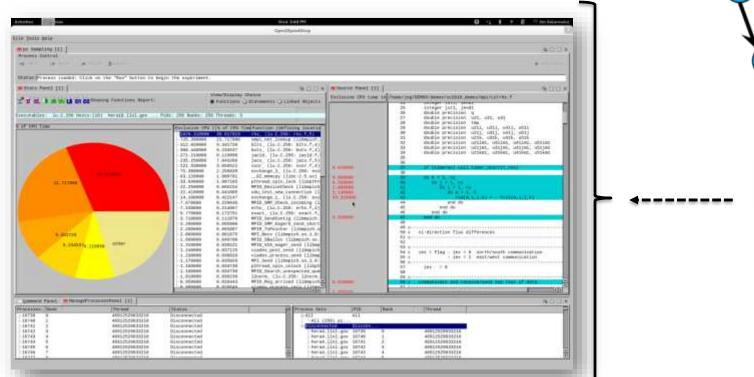
Open | SpeedShop Workflow

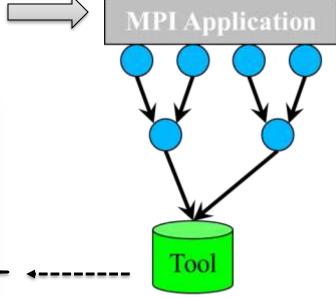




mpiexec_mpt -np 4 smg2000 -n 65 65 65

osspcsamp "mpiexec_mpt -np 4 smg2000 -n 65 65 65"





http://www.openspeedshop.org/

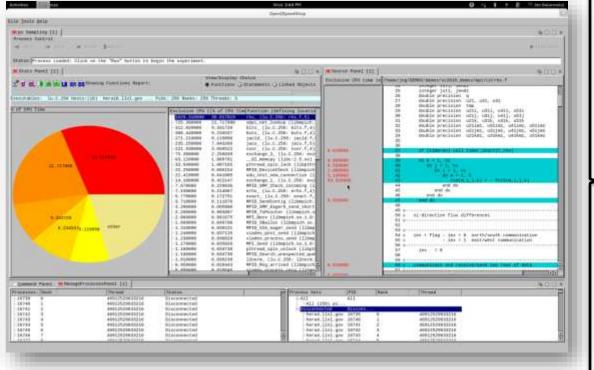
Open | SpeedShop Workflow

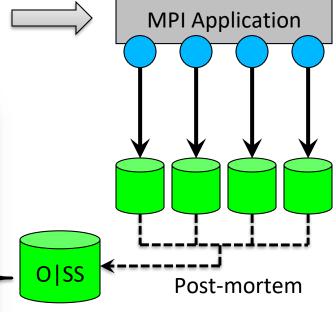




mpiexec_mpt -np 4 smg2000 -n 65 65 65

osspcsamp --offline "mpiexec_mpt -np 4 smg2000 -n 65 65 65"





http://www.openspeedshop.org/

Central Concept: Experiments





Users pick experiments:

- What to measure and from which sources?
- How to select, view, and analyze the resulting data?

Two main classes of performance data collection:

- Statistical Sampling
 - Periodically interrupt execution and record location
 - Useful to get an overview
 - Low and uniform overhead
- Event Tracing
 - Gather and store individual application events
 - Provides detailed per event information
 - Can lead to huge data volumes

O|SS can be extended with additional experiments

Sampling Experiments in O|SS





PC Sampling (pcsamp)

- Record PC repeatedly at user defined time interval
- > Low overhead overview of time distribution
- Good first step, lightweight overview

Call Path Profiling (usertime)

- > PC Sampling and Call stacks for each sample
- Provides inclusive and exclusive timing data
- Use to find hot call paths, caller and callee relationships

Hardware Counters (hwc, hwctime, hwcsamp)

- Provides profile of hardware counter events like cache & TLB misses
- hwcsamp:
 - Periodically sample to capture profile of the code against the chosen counter
 - Default events are PAPI_TOT_INS and PAPI_TOT_CYC
- hwc, hwctime:
 - Sample a hardware counter till a certain number of events (called threshold) is recorded and get Call Stack
 - Default event is PAPI_TOT_CYC

Tracing Experiments in O|SS





Input/Output Tracing (io, iot, iop)

- Record invocation of all POSIX I/O events
- Provides aggregate and individual timings
- Store function arguments and return code for each call (iot)
- Lightweight I/O profiling because not tracking individual call details (iop)

MPI Tracing (mpi, mpit, mpip)

- Record invocation of all MPI routines
- Provides aggregate and individual timings
- Store function arguments and return code for each call (mpit)
- Lightweight MPI profiling because not tracking individual call details (mpip)

Tracing Experiments in O|SS





Memory Tracing (mem)

- > 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
 - The total allocation
 - The total time spent in the call path
 - The start time for the first call

Additional Experiments in OSS/CBTF





CUDA NVIDIA GPU Event Tracing (cuda)

- > Record CUDA events, provides timeline and event timings
- > Traces all NVIDIA CUDA kernel executions and the data transfers between main memory and the GPU.
- Records the call sites, time spent, and data transfer sizes.

POSIX thread tracing (pthreads)

- > Record invocation of all POSIX thread events
- Provides aggregate and individual rank, thread, or process timings

OpenMP specific profiling/tracing (omptp)

Report task idle, barrier, and barrier wait times per OpenMP thread and attribute those times to the OpenMP parallel regions.

Performance Analysis in Parallel





How to deal with concurrency?

- > Any experiment can be applied to parallel application
 - Important step: aggregation or selection of data
- > Special experiments targeting parallelism/synchronization

❖ O SS supports MPI and threaded codes

- > Automatically applied to all tasks/threads
- > Default views aggregate across all tasks/threads
- > Data from individual tasks/threads available
- > Thread support (incl. OpenMP) based on POSIX threads

❖ Specific parallel experiments (e.g., MPI)

- Wraps MPI calls and reports
 - MPI routine time
 - MPI routine parameter information
- > The mpit experiment also stores function arguments and return codes for each call

Overview of where time is being spent: pcsamp





Basic syntax:

osspcsamp "how you run your executable normally"

Examples:

osspcsamp "smg2000 –n 50 50 50" osspcsamp "smg2000 –n 50 50 50" low

Parameters

Sampling frequency (samples per second)
Alternative parameter: high (200) | low (50) | default (100)

Recommendation: compile code with -g to get statements!

Example Run with Output





❖ osspcsamp "mpirun –np 4 smg2000 –n 50 50 50" (1/2)

```
Bash>osspcsamp "mpirun -np 4 ./smg2000 -n 50 50 50"
[openss]: pcsamp experiment using the default sampling rate: "100".
Creating topology file for frontend host localhost
Generated topology file: ./cbtfAutoTopology
Running pcsamp collector.
Program: mpirun -np 4 ./smg2000 -n 50 50 50
Number of mrnet backends: 4
Topology file used: ./cbtfAutoTopology
executing mpi program: mpirun -np 4 cbtfrun --mpi --mrnet -c pcsamp ./smg2000 -n 50 50 50
Running with these driver parameters:
(nx, ny, nz) = (65, 65, 65)
     <SMG native output>
Final Relative Residual Norm = 1.774415e-07
All Threads are finished.
default view for ./smg2000-pcsamp-0.openss
[openss]: The restored experiment identifier is: -x 1
Performance data spans 2.257689 seconds from 2016/11/09 13:33:33 to 2016/11/09 13:33:35
```

Example Run with Output





❖ osspcsamp "mpirun –np 4 smg2000 –n 50 50 50" (2/2)

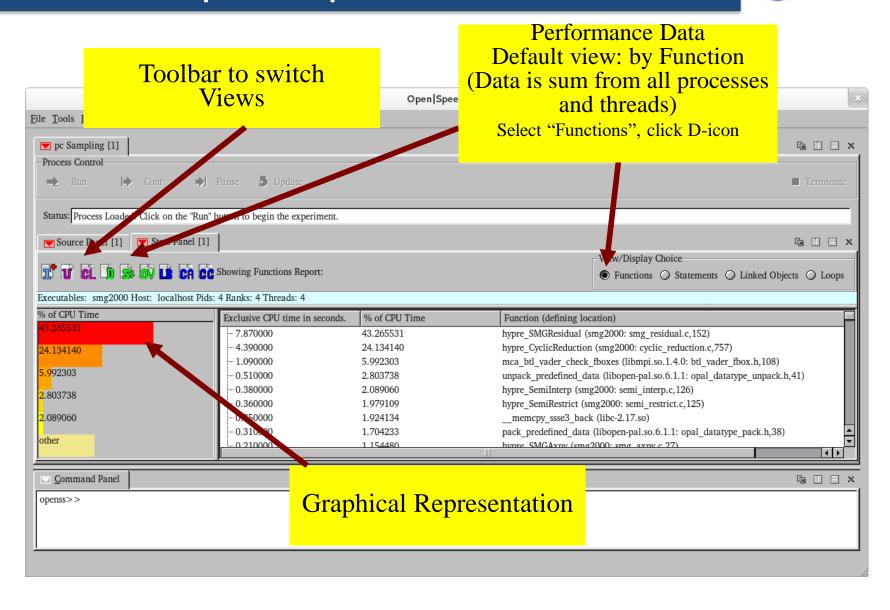
```
Exclusive % of CPU Function (defining location)
CPU time
   in
seconds.
2.850000 36.821705 hypre SMGResidual (smg2000: smg residual.c,152)
1.740000 22.480620 hypre CyclicReduction (smg2000: cyclic reduction.c,757)
0.410000 5.297158 mca btl vader check fboxes (libmpi.so.12.0.2: btl vader fbox.h,184)
0.250000 3.229974 opal progress (libopen-pal.so.13.0.2: opal progress.c,151)
0.250000 3.229974 hypre SemiInterp (smg2000: semi_interp.c,126)
0.190000 2.454780 pack predefined data (libopen-pal.so.13.0.2: opal_datatype_pack.h,35)
0.190000 2.454780 unpack predefined data (libopen-pal.so.13.0.2: opal_datatype_unpack.h,34)
0.120000 1.550388 int malloc (libc-2.17.so)
0.100000 1.291990 hypre SemiRestrict (smg2000: semi_restrict.c,125)
0.100000 1.291990 opal_generic_simple_pack (libopen-pal.so.13.0.2: opal_datatype_pack.c,274)
0.090000 1.162791 memcpy ssse3 back (libc-2.17.so)
0.080000 1.033592 int free (libc-2.17.so)
0.080000 1.033592 opal_generic_simple_unpack (libopen-pal.so.13.0.2:
opal datatype unpack.c,263)
```

❖ View with GUI: openss –f smg2000-pcsamp-0.openss

Default Output Report View



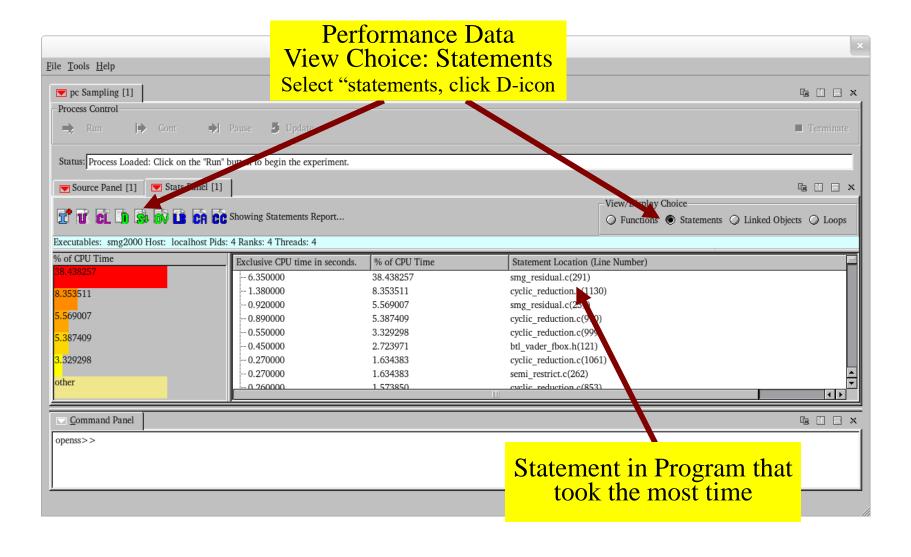




Statement Report Output View



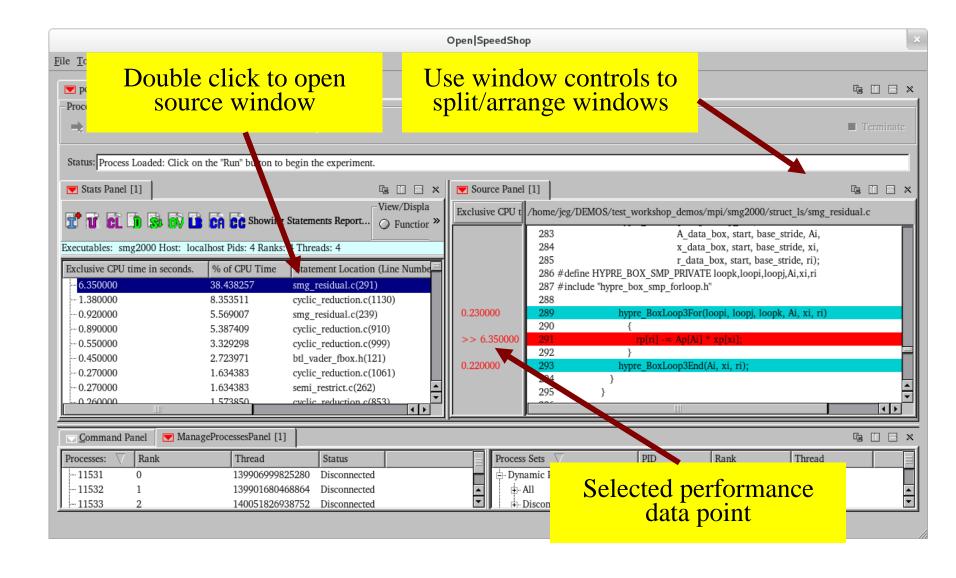




Associate Source & Performance Data



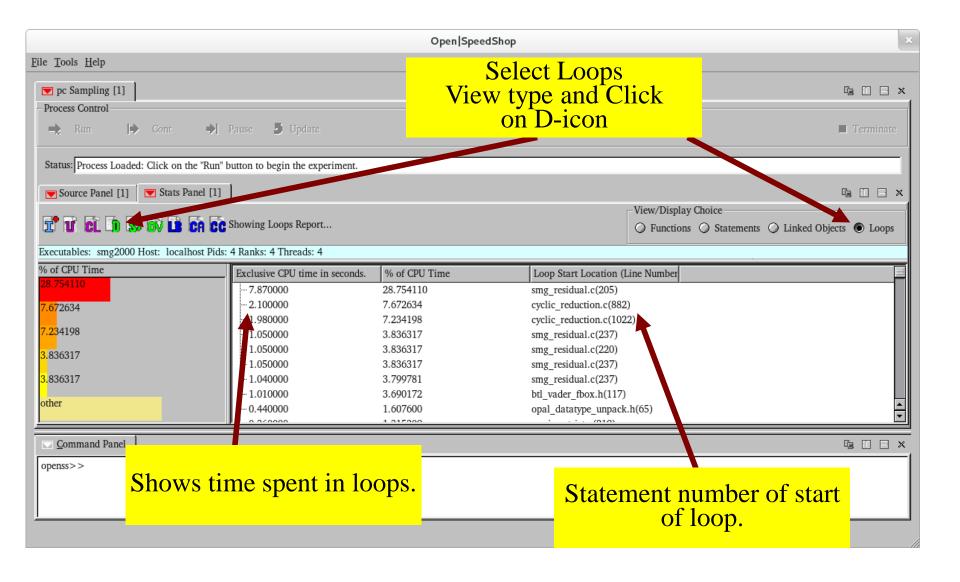




Loop View







Open | SpeedShop Basics





- Place the way you run your application normally in quotes and pass it as an argument to osspcsamp, or any of the other experiment convenience scripts: ossio, ossmpi, etc.
 - osspcsamp "mpiexec_mpt -np 64 ./mpi_application app_args"
- Open | SpeedShop sends a summary profile to stdout
- Open | SpeedShop creates a database file
- Display alternative views of the data with the GUI via:
 - > openss -f <database file>
- Display alternative views of the data with the CLI via:
 - > openss -cli -f <database file>
- Start with pcsamp for overview of performance
- Then, focus on performance issues with other experiments

Hands-on: Basic Sampling Experiment





Login Info and Hands-on exercise

- How to log into the tutorial computer system
 - Accounts on Pleiades are required for the NASA training exercises
 - Find these exercises (nasa_training_exercises.tar.gz) at:
 - www.openspeedshop.org/downloads
 - Unpack the nasa_training_exercises.tar.gz tarball into your \$HOME directory using:
 - tar –xzf nasa_training_exercises.tar.gz
- **❖** A file named EXERCISE is in the exercise directory that outlines the applications and exercises.
- The first exercise can be in this directory:
 - \$HOME/exercises/loop_check
- Consult README file in each of the directories for the instructions/guidance





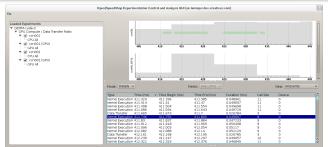
Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 2 Basic timing experiments and their Pros/Cons







Identifying Critical Regions





Flat Profile Overview

Profiles show computationally intensive code regions

> First views: Time spent per functions or per statements

Questions:

- Are those functions/statements expected?
- Do they match the computational kernels?
- Any runtime functions taking a lot of time?

Identify bottleneck components

- > View the profile aggregated by shared objects
- Correct/expected modules?
- > Impact of support and runtime libraries

Call path profiling & Comparisons





Call Path Profiling

- Take a sample: address inside a function
- > Call stack: series of program counter addresses (PCs)
- Unwinding the stack is walking through those address and recording that information for symbol resolution later.
- Leaf function is at the end of the call stack list

Open|SpeedShop: experiment called usertime

- > Time spent inside a routine vs. its children
- Key view: butterfly

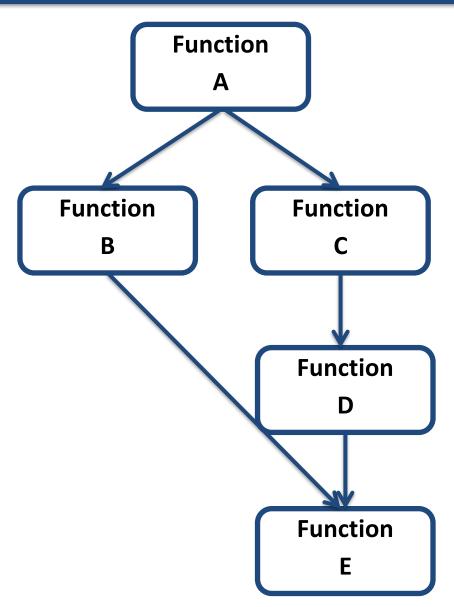
Comparisons

- > Between experiments to study improvements/changes
- > Between ranks/threads to understand differences/outliers

Adding Context through Stack Traces







Missing information in flat profiles

- Distinguish routines called from multiple callers
- Understand the call invocation history
- Context for performance data

Critical technique: Stack traces

- Gather stack trace for each performance sample
- Aggregate only samples with equal trace

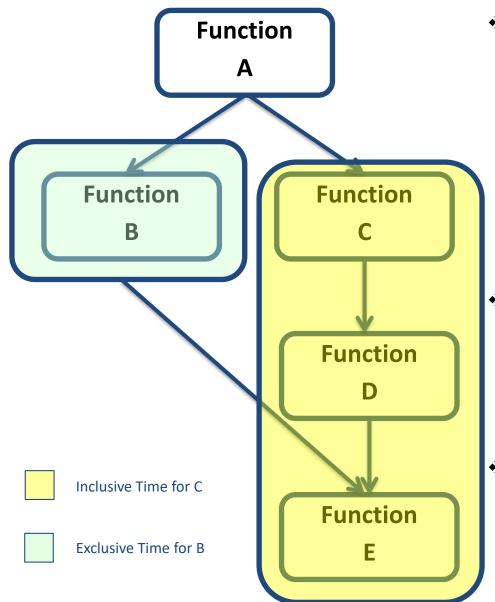
User perspective:

- Butterfly views (caller/callee relationships)
- > Hot call paths
 - Paths through application that take most time

Inclusive vs. Exclusive Timing







- Stack traces enable calculation of inclusive/exclusive times
 - Time spent inside a function only (exclusive)
 - See: Function B
 - Time spent inside a function and its children (inclusive)
 - See Function C and children
- Implementation similar to flat profiles
 - > Sample PC information
 - Additionally collect call stack information at every sample
- Tradeoffs
 - Pro: Obtain additional context information
 - Con: Higher overhead/lower sampling rate

Interpreting Call Context Data





Inclusive versus exclusive times

- > If similar: child executions are insignificant
 - May not be useful to profile below this layer
- > If inclusive time significantly greater than exclusive time:
 - Focus attention to the execution times of the children

Hotpath analysis

- Which paths takes the most time?
- > Path time might be ok/expected, but could point to a problem

Butterfly analysis (similar to gprof)

- > Should be done on "suspicious" functions
 - Functions with large execution time
 - Functions with large difference between inclusive and exclusive time
 - Functions of interest
 - Functions that "take unexpectedly long"
 - ...
- Shows split of time in callees and callers

Inclusive and Exclusive Time Profiles: Usertime





Basic syntax:

ossusertime "how you run your executable normally"

Examples:

ossusertime "smg2000 –n 50 50 50" ossusertime "smg2000 –n 50 50 50" low

Parameters

Sampling frequency (samples per second)
Alternative parameter: high (70) | low (18) | default (35)

Recommendation: compile code with -g to get statements!

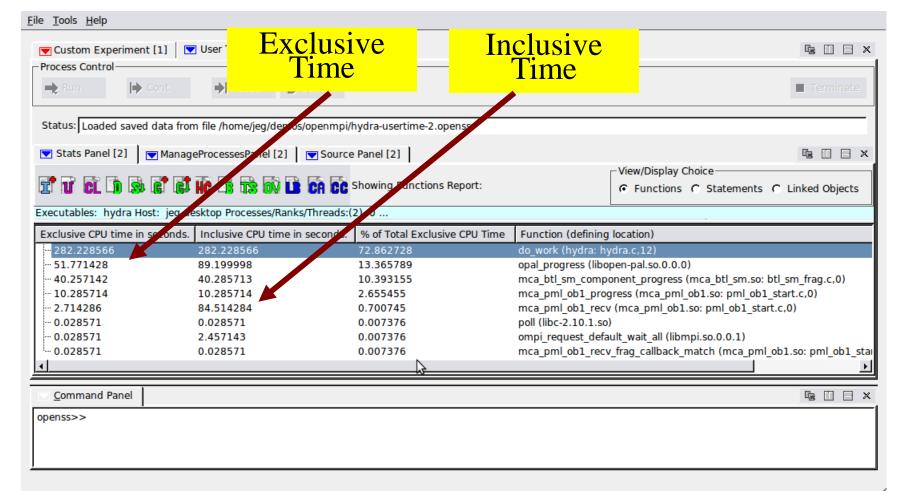
Reading Inclusive/Exclusive Timings





Default View

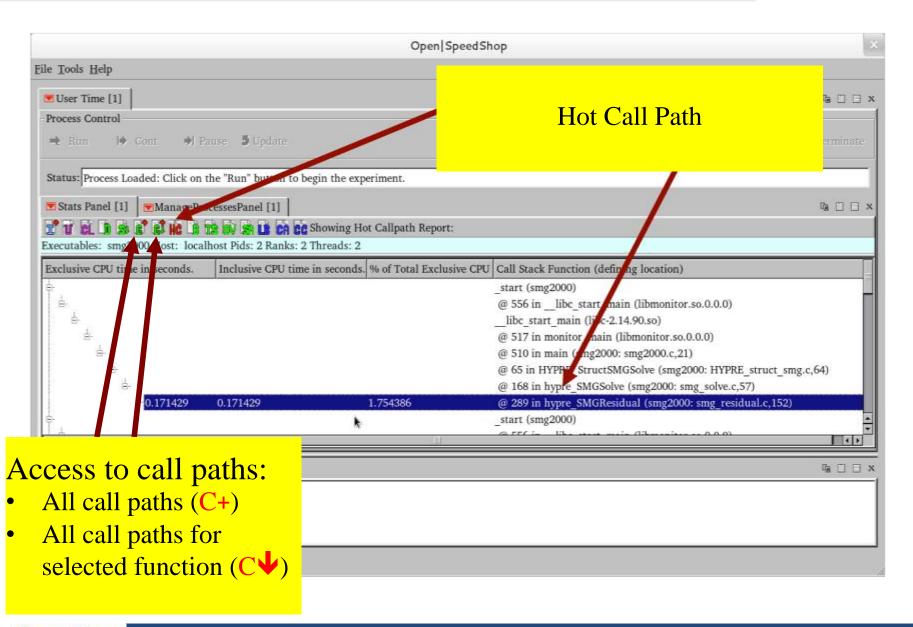
- Similar to pcsamp view from first example
- Calculates inclusive versus exclusive times



Stack Trace Views: Hot Call Path





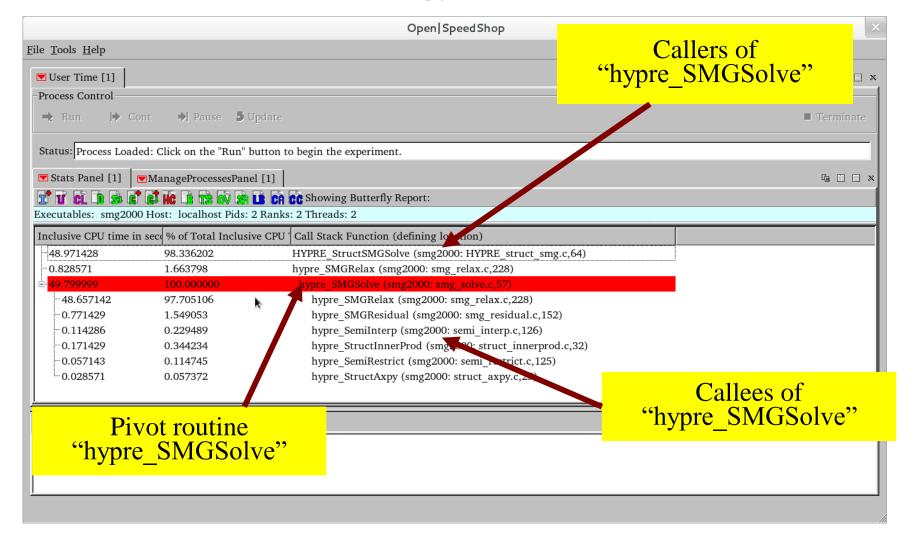


Stack Trace Views: Butterfly View





Similar to well known "gprof" tool



Comparing Performance Data





Key functionality for any performance analysis

- > Absolute numbers often don't help
- > Need some kind of baseline / number to compare against

Typical examples

- Before/after optimization
- Different configurations or inputs
- > Different ranks, processes or threads

Very limited support in most tools

- Manual operation after multiple runs
- Requires lining up profile data
- > Even harder for traces

Open|SpeedShop has support to line up profiles

- > Perform multiple experiments and create multiple databases
- Script to load all experiments and create multiple columns

Comparing Performance Data in O|SS





Convenience Script: osscompare

- Compares Open | SpeedShop up to 8 databases to each other
 - Syntax: osscompare "db1.openss,db2.openss,..." [options]
 - osscompare man page has more details
- Produces side-by-side comparison listing
- Data metric option parameter:
 - Compare based on: time, percent, a hwc counter, etc.
- Limit the number of lines by "rows=nn" option
- Specify the: viewtype=[functions|statements|linkedobjects]
 - Control the view granularity.
 - Compare based on the function, statement, or library level.
 - By default the compare will be done comparing the performance of functions in each of the databases.
 - If statements option is specified then all the comparisons will be made by looking at the performance of each statement in all the databases that are specified.
 - Similar for libraries, if linkedobject is selected as the viewtype parameter.

Comparison Report in O|SS





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 reduction.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:
topo unity component.c,0)
    0.280000000
                      0.210000000 hypre SemiInterp (smg2000: semi_interp.c,126)
    0.280000000
                      0.04000000 mca pml ob1 progress (libmpi.so.0.0.2:
topo unity component.c,0)
```

Summary / Timing analysis





Typical starting point:

- > Flat profile
- > Aggregated information on where time is spent in a code
- > Low and uniform overhead when implemented as sampling

Adding context

- > From where was a routine called, which routine did it call
- > Enables the calculation of exclusive and inclusive timing
- > Technique: stack traces combined with sampling

Key analysis options

- > Hot call paths that contains most execution time
- Butterfly view to show relations to parents/children

Comparative analysis

- Absolute numbers often carry little meaning
- Need the correct baseline, then compare against that

Hands-on: Basic Sampling Experiment – Call path





- Basic sampling application exercise
 - Also comparing runs to each other
- The call path exercise can be in these directories:
 - \$HOME/exercises/seq_lulesh
 - \$HOME/exercises/seq_smg2000/test
- Consult README file in each of the directories for the instructions/guidance





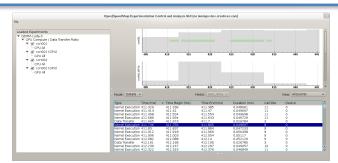
Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 3 Analysis of parallel codes: MPI, OpenMP, POSIX threads







Parallel Application Performance Challenges





Architectures are Complex and Evolving Rapidly

- > Changing multicore processor designs
- > Emergence of accelerators (GPGPU, MIC, etc.)
- Multi-level memory hierarchy
- > I/O storage sub-systems
- Increasing scale: number of processors, accelerators

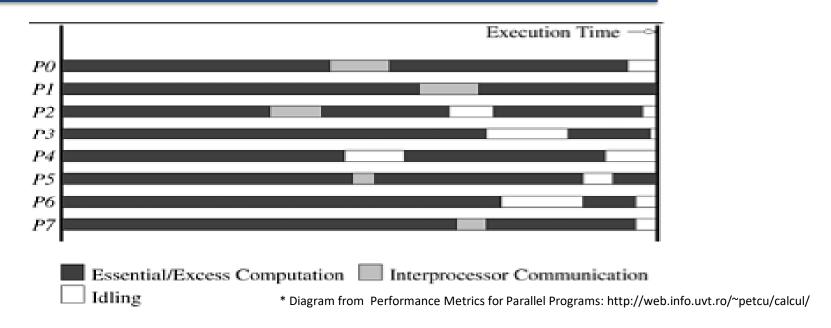
Parallel processing adds more performance factors

- > MPI communication time versus computation time
- > Threading synchronization time versus computation time
- > CPU time versus accelerator transfer and startup time tradeoffs
- I/O device multi-process contention issues
- > Efficient memory referencing across processes/threads
- Changes in application performance due to adapting to new architectures

Parallel Execution Goals







❖ Ideal scenario

- > Efficient threading when using pthreads or OpenMP
 - All threads are assigned work that can execute concurrently
 - Synchronization times are low.
- Load balance for parallel jobs using MPI
 - All MPI ranks doing same amount of work, so no MPI rank waits
- > Hybrid application with both MPI and threads
 - Limited amount of serial work per MPI process

Parallel Execution Goals





What causes the ideal goal to fail?

- > For MPI:
 - Equal work was not given to each rank
 - There is an out of balance communication pattern occurring
 - The application can't scale with the number of ranks being used
- > For threaded applications:
 - One or more threads doing more work than others and subsequently causing other threads to wait.
- > For hybrid applications:
 - Too much time spent between parallel/threaded regions
- > For multicore processors:
 - Remote memory references from the non-uniform access shared memory can cause sub-par performance
- > For accelerators:
 - Data transfers to the accelerator kernel might take more time than the speed-up for the accelerator operations on that data - also - is the CPU fully utilized?

Parallel Application Analysis Techniques





What steps can we take to analyze parallel jobs?

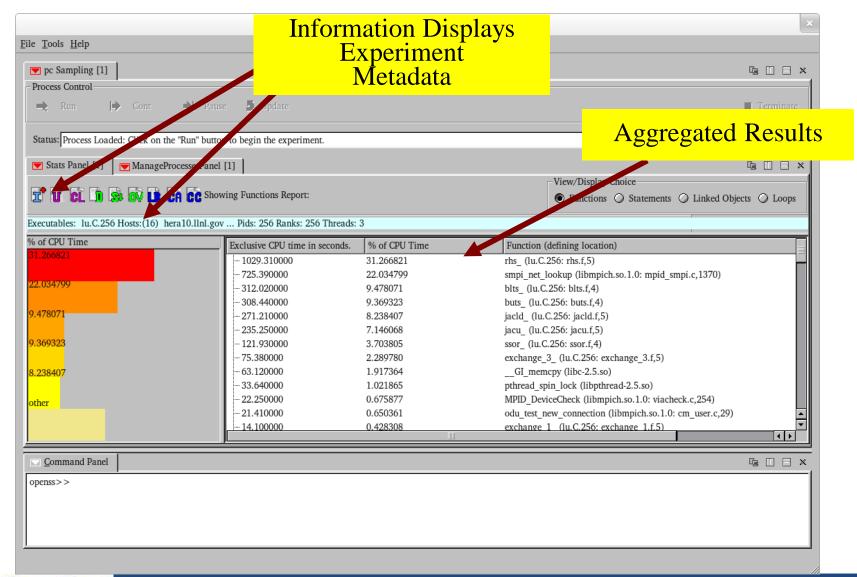
- Get an overview of where the time is being spent.
 - Use sampling to get a low overhead overview of time spent
 - Program counter, call stack, hardware counter
- > Examine overview information for all ranks, threads, ...
 - Analyze load balance information:
 - Min, max, and average values across the ranks and/or threads
 - Look at this information per library as well
 - Too much time in MPI could indicate load balance issue.
 - Use above info to determine if the program is well balanced
 - Are the minimum, maximum values widely different? If so:
 - Probably have load imbalance and need to look for the cause of performance lost because of the imbalance.
 - Not all ranks or threads doing the same amount of work
 - Too much waiting at barriers or synchronous global operations like MPI_Allreduce

pcsamp Default View: NPB: LU





Default Aggregated pcsamp Experiment View



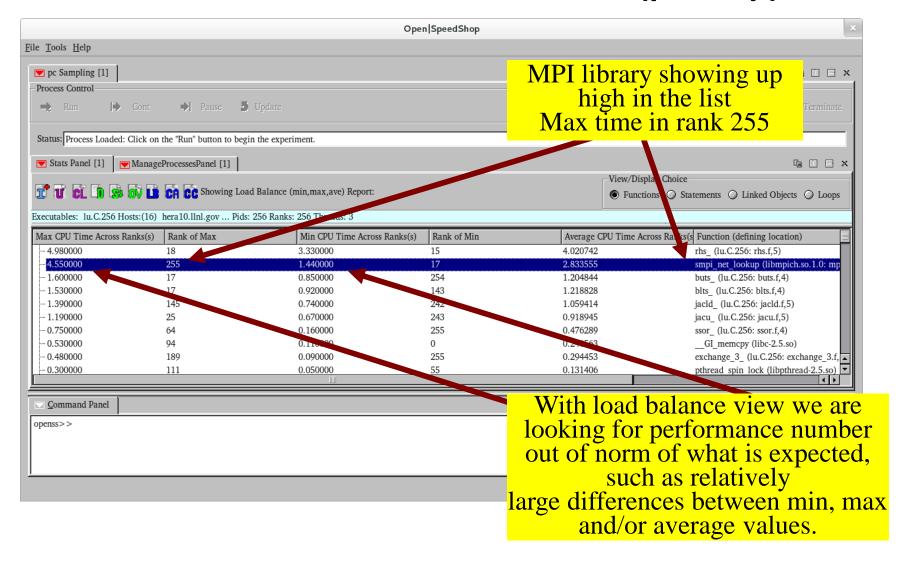
Dec 12, 2016

Load Balance View: NPB: LU





Load Balance View based on functions (pcsamp)



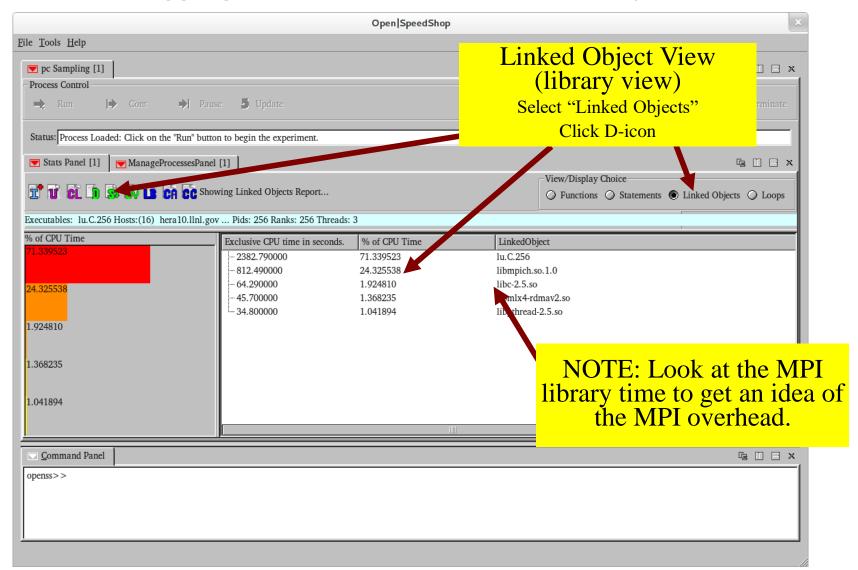
Dec 12, 2016

Default Linked Object View: NPB: LU





Default Aggregated View based on Linked Objects (libraries)



Parallel Execution Analysis Techniques





If imbalance detected, then what? How do you find the cause?

- Look at library time distribution across all the ranks, threads
 - Is the MPI library taking a disproportionate amount of time?
- ➤ If threaded (e.g. OpenMP), then look at the balance of time across worker threads.
 - For OpenMP look at idleness, barrier time, in addition to task times
- If MPI application, use a tool that provides per MPI function call timings
 - Can look at MPI function time distributions
 - In particular, MPI_Waitall
 - Then look at the call path to MPI_Waitall
 - Also, can look source code relative to
 - MPI rank or particular pthread that is involved.
 - Is there any special processing for the particular rank or thread
 - Examine the call paths and check code along path
- Use Cluster Analysis type feature, if tool has this capability
 - Cluster analysis can categorize threads or ranks that have similar performance into groups identifying the outlier rank or thread

Hot Call Paths View (CLI): NPB: LU





Hot Call Paths for MPI_Wait for rank 255 only

openss -cli -f lu-mpi-256.openss Show all call paths involving MPI_Wait openss>>expview -r 255 -vcalltrees,fullstack -f MPI Wait for rank 255 only % of Total Number of Calls Call Stack Function (defining locat **Exclusive MPI Call** Time(ms) >>>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) 3.878405 6010.978000 250 >>>>> @ 51 in MPI Wait (libmpich.so.1.0: wait.c,51) >>>main (lu.C.256) >>>> @ 140 in MAIN (lu.C.256: lu.f,46) Most expensive call >>>> @ 180 in ssor (lu.C.256: ssor.f,4) path to MPI_Wait >>>>> @ 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) 2798.770000 1.805823

Identifying Load Imbalance With O|SS





Get overview of application

- > Run a lightweight experiment to verify performance expectations
 - pcsamp, usertime, hwc

Use load balance view on pcsamp, usertime, hwc

- > Look for performance values outside of norm
 - Somewhat large difference for the min, max, average values
 - If the MPI libraries are showing up in the load balance for pcsamp, then do an MPI specific experiment

Use load balance view on MPI experiment

- > Look for performance values outside of norm
 - Somewhat large difference for the min, max, average values
- Focus on the MPI_Functions to find potential problems

Use load balance view on OpenMP experiment (omptp)

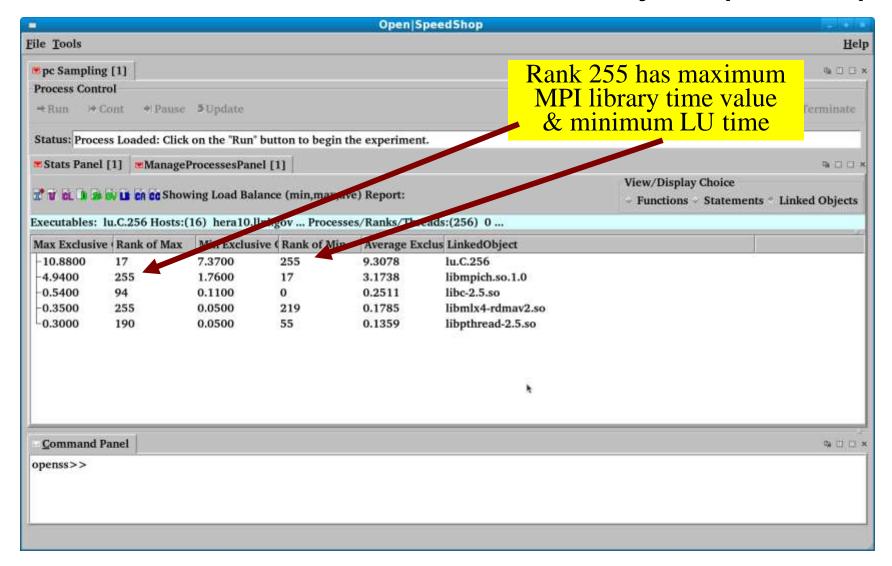
> Can also use expcompare across OpenMP threads

Link. Obj. Load Balance: Using NPB: LU





Load Balance View based on Linked Objects (libraries)



Using Cluster Analysis in O|SS





Can use with pcsamp, usertime, hwc

- Will group like performing ranks/threads into groups
- Groups may identify outlier groups of ranks/threads
- > Can examine the performance of a member of the outlier group
- Can compare that member with member of acceptable performing group

Can use with mpi, mpit, mpip

- > Same functionality as above
- But, now focuses on the performance of individual MPI Functions.
- Key functions are MPI_Wait, MPI_WaitAll
- > Can look at call paths to the key functions to analyze why they are being called to find performance issues

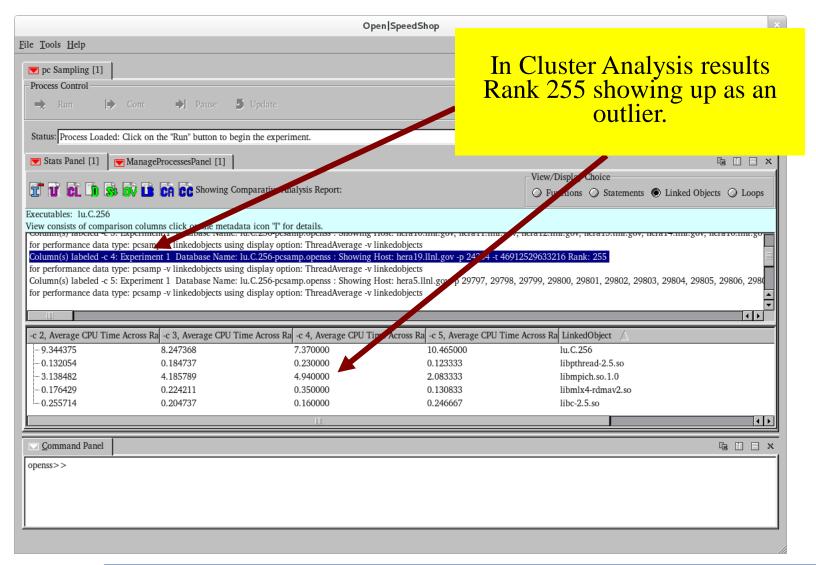
50

Link. Obj. Cluster Analysis: NPB: LU





Cluster Analysis View based on Linked Objects (libraries)



MPI/OpenMP Specific Experiments





MPI specific experiments

- > Record all MPI call invocations
- MPI functions are profiled (ossmpip)
 - Show call paths for each MPI unique call path, but individual call information is not recorded.
 - Less overhead than mpi, mpit.
- MPI functions are traced (ossmpi)
 - Record call times and call paths for each event
- MPI functions are traced with details (ossmpit)
 - Record call times, call paths and argument info for each event

OpenMP specific experiment (ossomptp)

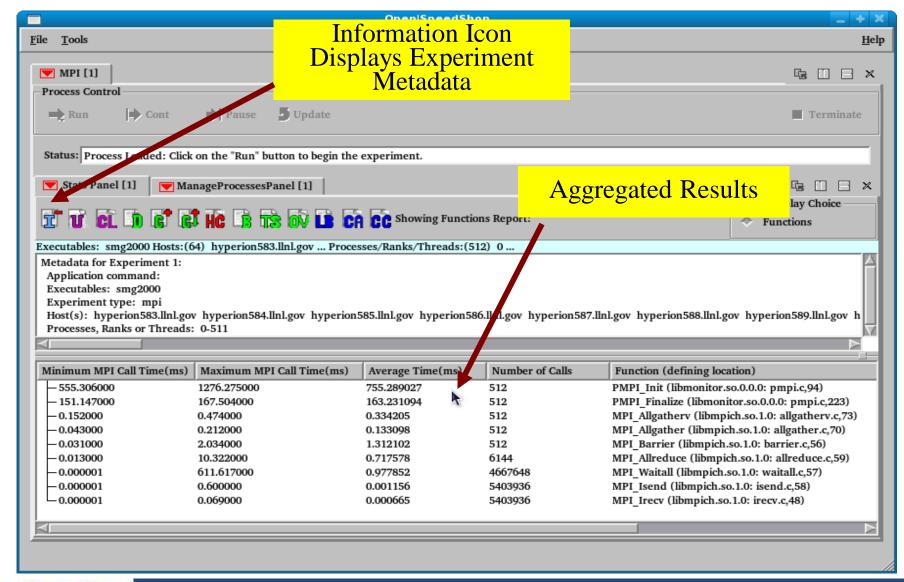
- Uses OMPT API to record task time, idleness, barrier, and wait barrier per OpenMP parallel region
 - Shows load balance for time
 - expcompare time across all threads

MPI Tracing Results: Default View





Default Aggregated MPI Experiment View



Using OMPTP experiment in O | SS

openss -cli -f ./matmult-omptp-0.openss





- The following three CLI examples show the most important ways to view OMPTP experiment data.
- Default view shows the timing of the parallel regions, idle, barrier, and wait barrier as an aggregate across all threads

```
openss>>expview
Exclusive Inclusive
                    % of Function (defining location)
times in times in
                   Total
seconds. seconds. Exclusive
           CPU Time
44.638794 45.255843 93.499987 compute. omp fn.1 (matmult: matmult.c,68)
 1.744841 1.775104 3.654726 compute_interchange._omp_fn.3 (matmult: matmult.c,118)
 0.701720 0.701726
                      1.469817 compute_triangular._omp_fn.2 (matmult: matmult.c,95)
                      1.366591 IDLE (omptp-collector-monitor-mrnet.so: collector.c,573)
0.652438 0.652438
0.004206 0.009359
                     0.008810 initialize. omp fn.0 (matmult: matmult.c,32)
                      0.000068 BARRIER (omptp-collector-monitor-mrnet.so: collector.c,587)
 0.000032
           0.000032
0.000000
           0.000000
                      0.000001 WAIT BARRIER (omptp-collector-monitor-mrnet.so: collector.c,602)
```

Using OMPTP experiment in O|SS





This example shows the comparison of exclusive time across all threads for the parallel regions, idle, barrier, and wait barrier

openss>>expcompare -mtime -t0:4

```
-t 2, -t 3, -t 4, Function (defining location)
Exclusive Exclusive Exclusive
times in times in times in
seconds, seconds, seconds,
11.313892 11.081346 11.313889 10.929668 compute._omp_fn.1 (matmult: matmult.c,68)
0.443713 0.430553
                     0.429635
                               0.440940 compute_interchange._omp_fn.3 (matmult: matmult.c,118)
0.253632
          0.213238
                     0.164875
                               0.069975 compute triangular. omp fn.2 (matmult: matmult.c,95)
 0.001047
           0.001100
                     0.001095
                                0.000964 initialize. omp fn.0 (matmult: matmult.c,32)
0.000008
          0.000008
                     0.000006
                               0.000010 BARRIER (omptp-collector-monitor-mrnet.so: collector.c,587)
0.000000
           0.000000
                     0.000000
                                0.00000 WAIT BARRIER (omptp-collector-monitor-mrnet.so: collector.c,602)
 0.000000
           0.247592
                     0.015956
                                0.388890 IDLE (omptp-collector-monitor-mrnet.so: collector.c,573)
```

Using OMPTP experiment in O|SS





This example shows the load balance of time across all threads for the parallel regions, idle, barrier, and wait barrier

openss>>expview -mloadbalance

Max O _l	penl	Mp Min	Open	Mp Average Function (defining location)
Exclusive Th	reac	dId Exclusive	Threa	adId Exclusive
Time Across	of M	ax Time Acro	ss of I	Min Time Across
OpenMp		OpenMp		
ThreadIds(s)		ThreadIds(s)		ThreadIds(s)
11.313892	0	10.929668	4	11.159699 computeomp_fn.1 (matmult: matmult.c,68)
0.443713	0	0.429635	3	0.436210 compute_interchangeomp_fn.3 (matmult: matmult.c,118)
0.388890	4	0.015956	3	0.217479 IDLE (omptp-collector-monitor-mrnet.so:
collector.c,573)				
0.253632	0	0.069975	4	0.175430 compute_triangularomp_fn.2 (matmult: matmult.c,95)
0.001100	2	0.000964	4	0.001052 initializeomp_fn.0 (matmult: matmult.c,32)
0.000010	4	0.000006	3	0.000008 BARRIER (omptp-collector-monitor-mrnet.so:
collector.c,587	')			
0.000000	0	0.000000	0	0.000000 WAIT_BARRIER (omptp-collector-monitor-mrnet.so:
collector.c,602	2)			

Summary / Parallel Bottlenecks





Open|SpeedShop supports MPI, OpenMP, and threaded applications (including hybrid)

> Works with multiple MPI implementations

Parallel experiments

- > Apply the sequential O|SS collectors to all nodes
- > Specialized MPI profiling and tracing experiments
- Specialized OpenMP profiling experiment

Result Viewing

- > Results are aggregated across ranks/processes/threads
- Optionally: select individual ranks/threads or groups
- Specialized views:
 - Load balance view
 - Cluster analysis
- Use features to isolate sections of problem code

Hands-on Section 3: Going Parallel - threading





- Parallel related parallel application exercise (threading)
- Threading exercises can be found in these directories:
 - > \$HOME/exercises/matmul
 - > Supplemental:
 - \$HOME/exercises/lulesh2.0.3
- Parallel related application exercise (MPI)
- MPI exercises can be found in these exercise directories:
 - \$HOME/exercises/mpi_nbody
 - Supplemental:
 - \$HOME/exercises/smg2000/test
- Consult README file in each of the directories for the instructions/guidance





Open SpeedShop™

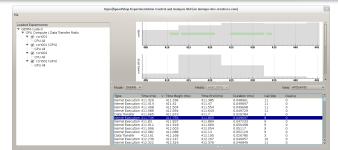


NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 4

Advanced analysis: Hardware Counter Experiments







Identify architectural impact on code inefficiencies





Timing information shows where you spend your time

- > Hot functions / statements / libraries
- Hot call paths

BUT: It doesn't show you why

- > Are the computationally intensive parts efficient?
- > Are the processor architectural components working optimally?

Answer can be very platform dependent

- Bottlenecks may differ
- > Cause of missing performance portability
- Need to tune to architectural parameters

Next: Investigate hardware/application interaction

- Efficient use of hardware resources or Micro-architectural tuning
- > Architectural units (on/off chip) that are stressed

Good Primary Focus: Efficient movement of data





Modern memory systems are complex

- Deep hierarchies
- Explicitly managed memory
- > NUMA behavior
- Streaming/Prefetching

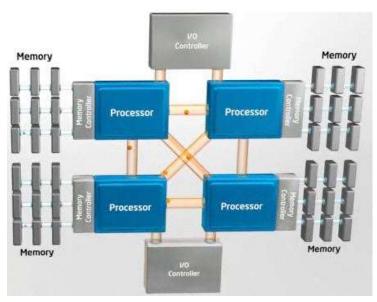
Data Location	Access Latency, ns (Sandy Bridge, 2.6GHZ)
L1	1.2
L2	3.5
L3	6.5
DRAM	28

Key to performance: Data locality and Concurrency

- Accessing the same data repeatedly(Temporal)
- Accessing neighboring data(Spatial)
- Effective/parallel use of cores

Information to look for

- Load/Store Latencies
- Prefetch efficiency
- Cache miss rate at all levels
- > TLB miss rates
- NUMA overheads



Another important focus: Efficient Vectorization





Newer processors have wide vector registers

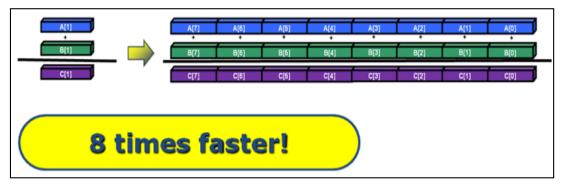
- Intel Xeon 2670, Sandy Bridge: 256 bits floating point registers, AVX (8 Real / 4 Double)
- Intel Xeon Phi, Knights Corner: 512 bits(16 Real / 8 Double)
- Intel Haswell 256 bits Integer Registers, AVX2 : FMA (2X the peak flops)

Key to performance: Vectorization

- Compiler Vectorization
- Use of 'intrinsics'
- Use of Pragmas to help the compiler
- Assembly code

Analysis Options

- Compiler vectorization report
- Look at assembly code
- Measure performance with PAPI counters



Going from Scalar to Intel® AVX can provide up to 8x faster performance

Hardware Performance Counters





Architectural Features

- Typically/Mostly packaged inside the CPU
- Count hardware events transparently without overhead

Newer platforms also provide system counters

- Network cards and switches
- > Environmental sensors

Drawbacks

- > Availability differs between platform & processors
- > Slight semantic differences between platforms
- > In some cases : requires privileged access & kernel patches

* Recommended: Access through PAPI

- > API for tools + simple runtime tools
- > Abstractions for system specific layers
- More information: http://icl.cs.utk.edu/papi/

The O|SS HWC Experiments





Provides access to hardware counters

- > Implemented on top of PAPI
- Access to PAPI and native counters
- > Examples: cache misses, TLB misses, bus accesses

Basic model 1: Timer Based Sampling: HWCsamp

- > Samples at set sampling rate for the chosen event
- Supports multiple counters
- Lower statistical accuracy
- Can be used to estimate good threshold for hwc/hwctime

Basic model 2: Thresholding: HWC and HWCtime

- User selects one counter
- > Run until a fixed number of events have been reached
- > Take PC sample at that location
 - HWCtime also records stacktrace
- Reset number of events
- > Ideal number of events (threshold) depends on application

Examples of Typical Counters (Xeon E5-2670)





PAPI Name	Description	Threshold
PAPI_L1_DCM	L1 data cache misses	high
PAPI_L2_DCM	L2 data cache misses	high/medium
PAPI_L3_TCM	L3 cache misses	high
PAPI_TOT_INS	Instructions completed	high
PAPI_STL_ICY	Cycles with no instruction issue	high/medium
PAPI_BR_MSP	Miss-predicted branches	medium/low
PAPI_DP_OPS	Number of 64-Bit floating point Vector OPS	high
PAPI_LD_INS	Number of load instructions	high
PAPI_VEC_DP	Number of vector/SIMD instructions – 64Bit	high
PAPI_BR_INS	Number of branch instructions	low
PAPI_TLB_TL	Number of TLB misses	low

Note: Threshold indications are just rough guidance and depend on the application.

Note: counters platform dependent (use papi avail & papi native avail)

Suggestions to Manage Complexity





- The number of PAPI counters and their use can be overwhelming; Some guidance here with a few "Metric-Ratios".
 - 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
- This presentation will illustrate with some examples of the use of Metric-Ratios

How to use OSS HWCsamp experiment





- - > Sequential job example:
 - osshwcsamp "smg2000"
 - Parallel job example:
 - osshwcsamp "mpirun –np 128 smg2000 –n 50 50 50"
 PAPI_L1_DCM,PAPI_L1_TCA 50
- default events: PAPI_TOT_CYC and PAPI_TOT_INS
- default sampling_rate: 100
- <PAPI_event_list>: Comma separated PAPI event list (Maximum of 6 events that can be combined)
- <sampling_rate>:Integer value sampling rate
- Use event count values to guide selection of thresholds for HWC, HWCtime experiments for deeper analysis

Selecting the Counters & Sampling Rate





For osshwcsamp, Open | SpeedShop supports ...

- Derived and Non derived PAPI presets
 - All derived and non derived events reported by "papi avail"
 - Also reported by running "osshwcsamp" with no arguments
 - Ability to sample up to six (6) counters at one time; before use test with
 papi_event_chooser PRESET < list of events>
 - If a counter does not appear in the output, there may be a conflict in the hardware counters
- > All native events
 - Architecture specific (incl. naming)
 - Names listed in the PAPI documentation
 - Native events reported by "papi native avail"

Sampling rate depends on application

- > Overhead vs. Accuracy
 - Lower sampling rate causes less samples

Useful Metric-Ratio 1: IPC





- Instructions Per Cycle(IPC) also referred to as Computational Intensity
 - > IPC= PAPI_TOT_INS/PAPI_TOT_CYCLES
- Data from single-core Xeon E5-2670, Sandy Bridge
- In the table below compiler optimization -O1 used to bring out differences in IPC based on stride used with different loop order;
- If you use -O2 for this simple case compiler does the right transformations, permuting loop order and vectorizing to yield IPC = 3.594 (jki order); This improves access to memory through cache.
- Importance of stride through the data is illustrated with this simple example; Compiler may not always do the needed optimization. Use IPC values from functions and loops to understand efficiency of data access through your data structures.

- Example matrix multiply;Triple do loop;(n1=n2=n3=1000)
- code for loop order 'ijk'; All vectors 'double'

Metric	IJK	IKJ JIK JKI		KIJ	КЛ	MATMUL	DGEMM	
PAPI_TOT_INS	8.012E+09	9.011E+09	8.011E+09	9.01E+09	9.01E+09	9.011E+09	9.016E+09	7.405E+08
PAPI_TOT_CYC	2.42E+10	5.615E+10	2.423E+10	2.507E+09	5.612E+10	2.61E+09	2.601E+09	2.859E+08
IPC	0.331	0.160	0.331	3.594	0.161	3.452	3.466	2.590
MFLOPS	272	117	271	2625	117	2525	2532	19233 (93% peak)

Useful Metric-Ratio 2: FloatOps/Cycle





- Traditionally PAPI_FP_INS/PAPI_TOT_CYC used to evaluate relative floating point density
 - For a number of reasons measuring and analyzing floating point performance on Intel Sandy Bridge and Ivy bridge must be done with care. See PAPI web site for full discussion. The reasons are: instruction mix scalar instructions + vector (AVX, SSE) packed instructions, hyperthreading, turbo-mode and speculative execution.
 - > The floating point counters have been disabled in the newer Intel Haswell cpu architecture
 - > On Sandy Bridge and Ivy Bridge PAPI_FP_INS is no longer an appropriate counter if loops are vectorized
 - > No single PAPI metric captures all floating point operations
- We provide some guidance with useful PAPI Preset counters. Data from single-core Xeon E5-2670, Sandy Bridge. Double precision array operations for Blas1(daxpy), Blas2(dgemv) and Blas3(dgemm) are benchmarked. Matrix size=nxn; vector size=nx1. Data array sizes are picked to force operations from DRAM memory
- Table below shows measured PAPI counter data for a few counters and compares the measured FLOP/Ops against theoretical expectations.
- PAPI_DP_OPS and PAPI_VEC_DP give similar values and these counter values correlate well with expected floating point operation counts for double precision.

Blas Operation	n	Thererical mem refs or Ops	Theoretical FLOP	Theoretical FLOP/Ops	wall time,	тот сус	TOT INS	FP INS	LD INS	SR INS	DP OPS	PAPI GFLOPS	PAPI FLOP/Ops
daxpy	2.50E+07	7.5E+07	5.0E+07	0.67	0.03	1.04E+08	5.20E+07	11.52	2.50E+07	1.25E+07	5.01E+07	1.56	0.668
dgemv	1.00E+04	1.0E+08	2.0E+08	2	0.06073	2.16E+08	1.69E+08	29.12	6.25E+07	1.25E+07	2.36E+08	3.89	1.57557985
dgemm	1.00E+04	4.00E+08	2E+12	5000.00	80.937	2.67E+11	7.33E+11	7.2	1.12E+11	1.38E+09	2.01E+12	24.80	8.83518225

For Intel Haswell FloatOps not available: Use IPC or CPI





- We again provide some guidance with data from a single-core of a Haswell Processor (Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz)
- Blas1, Blas2 and Blas3 kernels as in the previous slide are benchmarked. Matrix size=nxn; vector size=nx1. Data array sizes are picked to force operations from DRAM memory
- Table below shows measured PAPI counter data for a few counters and metric ratio IPC
- When operating at peak performance, Haswell can retire 4 micro-ops/cycle

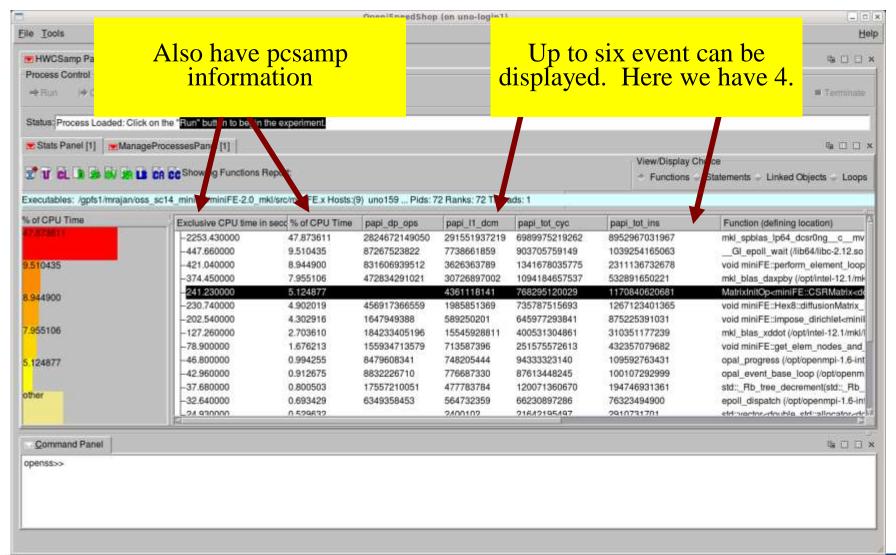
	Thererical			wall										
		Theoretica	Theoretical											
				•	TOT CYC	TOT INS	IPC		CPI		LD INS	SR INS	GFLOPS	FLOP/mem-Ops
						_					_	_		
2.50E+07	7.50E+07	5.00E+07	0.67	3.24E-02	1.17E+08	6.25E+07	1	0.54		1.87	3.13E+07	1.25E+07	1.53932	0.57
1.00E+04	1.00E+08	2.00E+08	2	6.11E-02	2.2E+08	2.06E+08	}	0.94		1.06	7.81E+07	1.25E+07	3.272	1.10
1.00E+04	4.00E+08	2.00E+12	5000	41.8546	1.38E+11	4.65E+11		3.36		0.30	1.9E+11	1.23E+09	47.7655	5.23

hwcsamp with miniFE (see mantevo.org)





- osshwcsamp "mpiexec –n 72 miniFE.X –nx 614 –ny 614 –nz 614" PAPI_DP_OPS,PAPI_L1_DCM,PAPI_TOT_CYC,PAPI_TOT_INS
- openss –f miniFE.x-hwcsamp.openss

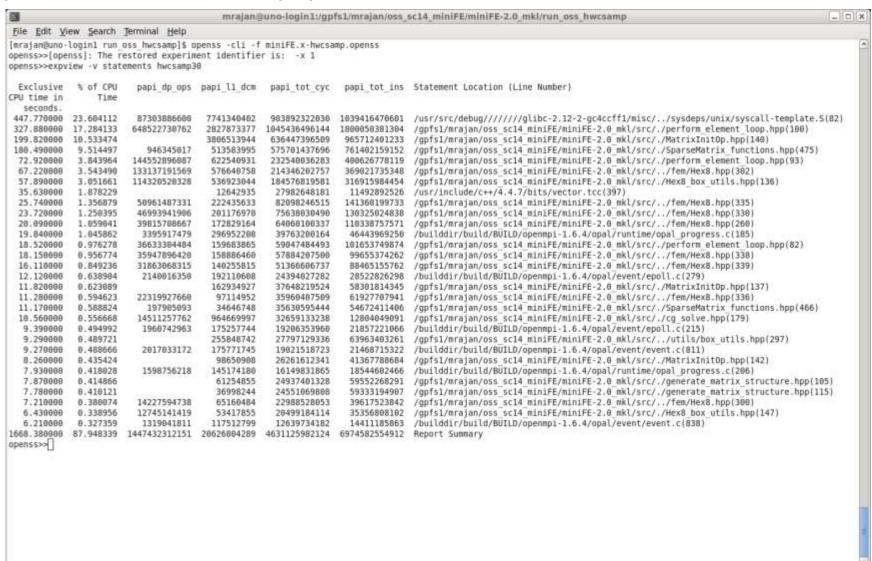


Viewing hwcsamp Data in CLI





openss -cli -f miniFE.x-hwcsamp.openss



Viewing Data in CLI





Selections of CLI commands used to view the data:

- expview -v linkedobjects
- ❖ expview –m loadbalance
- expview –v statements hwcsamp<number>
 - Example to show top 10 statements:
 - expview –v statements hwcsamp10
- expview –v calltrees,fullstack usertime<number>
- ❖ expcompare r 1 –r 2 –m time (compares rank 1 to rank 2 for metric equal time)

Deeper Analysis with HWC and HWCtime





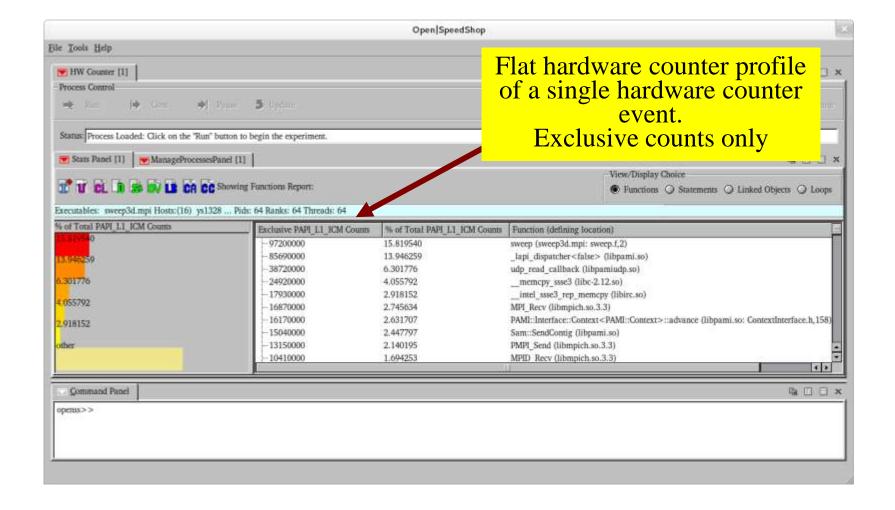
- osshwc[time] "<command> < args>" [default |
 <PAPI_event> | <PAPI threshold> | <PAPI_event><PAPI
 threshold>]
 - Sequential job example:
 - osshwc[time] "smg2000 –n 50 50 50" PAPI_FP_OPS 50000
 - Parallel job example:
 - osshwc[time] "mpirun –np 128 smg2000 –n 50 50 50"
- default: event (PAPI_TOT_CYC), threshold (10000000)
- <PAPI_event>: PAPI event name
- <PAPI threshold>: PAPI integer threshold
- ❖ NOTE: If the output is empty, try lowering the <threshold> value. There may not have been enough PAPI event occurrences to record and present

Viewing hwc Data





hwc default view: Counter = Instruction Cache Misses

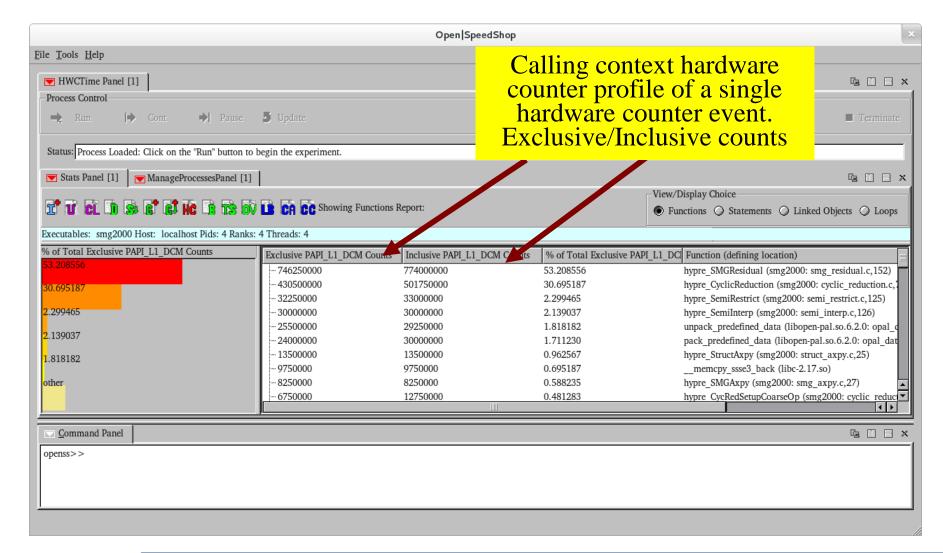


Viewing hwctime Data





hwctime default view: Counter = L1 Data Cache Misses



Example 1 on use of PAPI: LLNL Sparse Solver Benchmark AMG





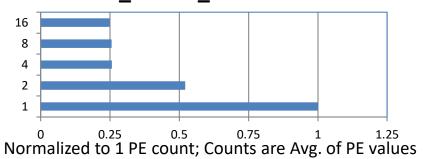
Major reasons on-node scaling limitations

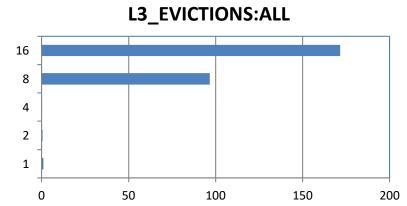
- Memory Bandwidth
- > Shared L3 Cache

L3 cache miss for 1,2,4 Pes matches expectation for strong scaling

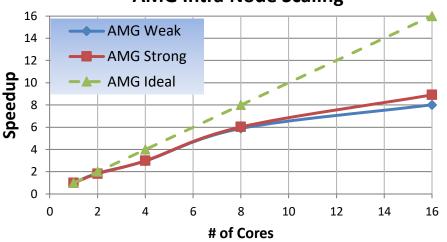
- Reduced data per PE
- > L3 misses decreasing up to 4 PEs linearly.

L3_CACHE_MISSES:ALL





AMG Intra Node Scaling



- On the other hand L3 Evictions for 1,2,4 PEs similarly decrease 'near-perfect' but dramatically increases to 100x at 8PEs and 170x at 16 PEs
- L3 evictions are a good measure of memory bandwidth limited performance bottleneck at a node
- General Memory BW limitation Remedies
 - > Blocking
 - Remove false sharing for threaded codes

Example 2 on use of PAPI: False Cache-line sharing in OpenMP





! Cache line **UnAligned**real*4, dimension(100,100)::c,d
!\$OMP PARALLEL DO
do i=1,100
do j=2, 100
c(i,j) = c(i, j-1) + d(i,j)
enddo
enddo
!\$OMP END PARALLEL DO

```
! Cache line Aligned
real*4, dimension(112,100)::c,d
!$OMP DO SCHEDULE(STATIC, 16)
do i=1,100
do j=2, 100
c(i,j) = c(i, j-1) + d(i,j)
enddo
enddo
!$OMP END DO
```

Same computation, but careful attention to alignment and independent OMP parallel cache-line chunks can have big impact; L3_EVICTIONS a good measure;

	Run Time	L3_EVICTIONS:ALL	L3_EVICTIONS:MODIFIED
Aligned	6.5e-03	9	3
UnAligned	2.4e-02	1583	1422
Perf. Penalty	3.7	175	474

Hands-on Section 4: Architectural Details





- Hardware counter experiments related exercises
- HWC exercises can be found in these directories:
 - \$HOME/exercises/soa_aos
 - \$HOME/exercises/matrix_multiply
 - > Supplemental exercises:
 - \$HOME/exercises/HPCCG-0.5
 - \$HOME/exercises/HPCCG-0.5_from_snl (no run just view)
- Consult README file in each of the directories for the instructions/guidance





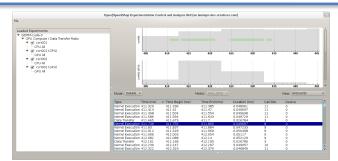
Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 5 Analysis of I/O







Need for Understanding I/O





❖ I/O could be significant percentage of execution time dependent upon:

- > Checkpoint, analysis output, visualization & I/O frequencies
- I/O pattern in the application:N-to-1, N-to-N; simultaneous writes or requests
- Nature of application: data intensive, traditional HPC, out-of-core
- File system and Striping: NFS, Lustre, Panasas, and # of Object Storage Targets (OSTs)
- ➤ I/O libraries: MPI-IO, hdf5, PLFS,...
- Other jobs stressing the I/O sub-systems

Obvious candidates to explore first while tuning:

- Use parallel file system
- Optimize for I/O pattern
- > Match checkpoint I/O frequency to MTBI of the system
- Use appropriate libraries

I/O Performance Example





Application: OOCORE benchmark from DOD HPCMO

- > Out-of-core SCALAPACK benchmark from UTK
- Can be configured to be disk I/O intensive
- Characterizes a very important class of HPC application involving the use of Method of Moments (MOM) formulation for investigating electromagnetics (e.g. Radar Cross Section, Antenna design)
- Solves dense matrix equations by LU, QR or Cholesky factorization
- "Benchmarking OOCORE, an Out-of-Core Matrix Solver," Cable, S.B., D'Avezedo, E. SCALAPACK Team, University of Tennessee at Knoxville/U.S. Army Engineering and Development Center

Why use this example?





- Used by HPCMO to evaluate I/O system scalability
- Out-of-core dense solver benchmarks demonstrate the importance of the following in performance analysis:
 - > I/O overhead minimization
 - ➤ Matrix Multiply kernel possible to achieve close to peak performance of the machine if tuned well
 - "Blocking" very important to tune for deep memory hierarchies

Use O|SS to measure and tune for I/O





INPUT: testdriver.in

ScaLAPACK out-of-core LU,QR,LL factorization input file

testdriver.out

_	
6	device out
U	acvice out

1 number of factorizations

LU factorization methods -- QR, LU,

or LT

1 number of problem sizes

31000 values of M

31000 values of N

1 values of nrhs

9200000 values of Asize

1 number of MB's and NB's

16 values of MB

values of NB

1 number of process grids

4 values of P

4 values of Q

Run on 16 cores on an SNL Quad-Core, Quad-Socket Opteron IB Cluster

Investigate File system impact with OpenSpeedShop: Compare Lustre I/O with striping to NFS I/O

run cmd: ossio "srun -N 1-n 16 ./testzdriver-std"

Sample Output from Lustre run:

TIME M N MB NB NRHS P Q Fact/SolveTime Error Residual

---- ------ ------ --- ---- -----

WALL 31000 31000 16 16 1 4 4 1842.20 1611.59 4.51E+15 1.45E+11

DEPS = 1.110223024625157E-016

 $sum(xsol_i) = (30999.9999999873, 0.0000000000000000E+000)$

sum |xsol_i - x_i| = (3.332285336962339E-006,0.000000000000000E+000)

From output of two separate runs using Lustre and NFS:

LU Fact time with Lustre= 1842 secs; LU Fact time with NFS = 2655 secs

813 sec penalty (more than 30%) if you do not use parallel file system like Lustre!

NFS and Lustre O | SS Analysis (screen shot from NFS)





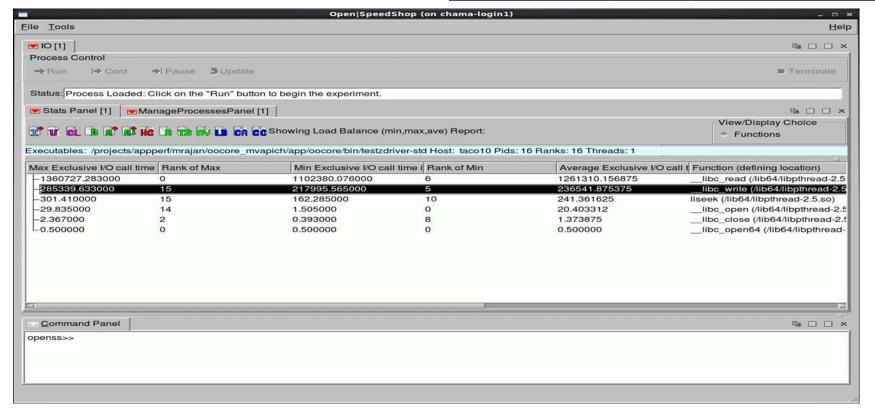
I/O to Lustre instead of NFS reduces runtime 25%: (1360 + 99) – (847 + 7) = 605 secs

NFS RUN

LUSTRE RUN

Min t (secs)	Max t (secs)	Avg t (secs)	call Function
			libc_read(/lib64/libpthread-
1102.380076	1360.727283	1261.310157	2.5.so)
			libc_write(/lib64/libpthread-
31.19218	99.444468	49.01867	2.5.so)

Min t (secs)	Max t (secs)	Avg t (secs)	call Function			
			libc_read(/lib64/libpthread-			
368.898283	847.919127	508.658604	2.5.so)			
			libc_write(/lib64/libpthread-			
6.27036	7.896153	6.850897	2.5.so)			



Lustre file system striping





Lustre File System (lfs) commands:

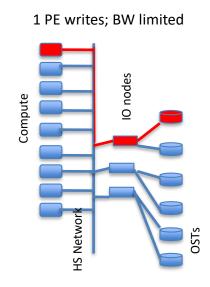
Ifs setstripe –s (size bytes; k, M, G) –c (count; -1 all) –I (index; -1 round robin) <file | directory>
Typical defaults: -s 1M -c 4 –i -1 (usually good to try first)

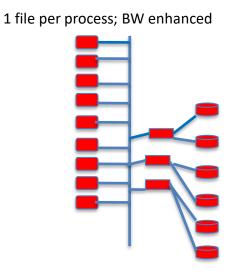
File striping is set upon file creation

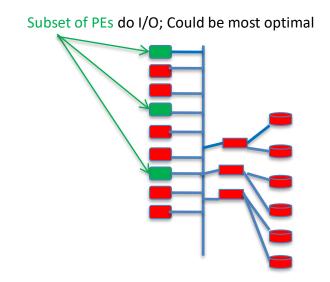
Ifs getstripe <file | directory>

Example: If s getstripe --verbose ./oss_lfs_stripe_16 | grep stripe_count

stripe_count: 16 stripe_size: 1048576 stripe_offset: -1



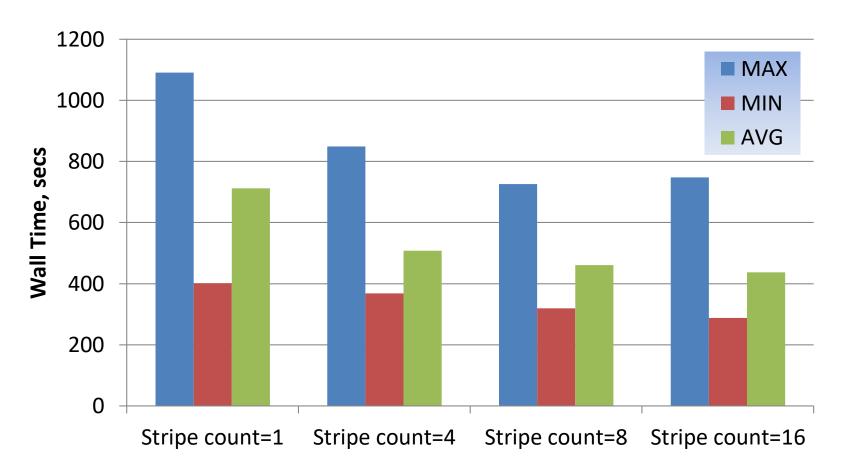








OOCORE I/O performance libc_read time from OpenSpeedShop



Additional I/O analysis with O|SS





Extended I/O Tracing (iot experiment)

- > Records each event in chronological order
- Collects Additional Information
 - Function Parameters
 - Function Return Value
- ➤ When to use extended I/O tracing?
 - When you want to trace the exact order of events
 - When you want to see the return values or bytes read or written.
 - When you want to see the parameters of the IO call

Beware of Serial I/O in applications: Encountered in VOSS, code LeP: Simple code here illustrates (acknowledgment: Mike Davis, Cray, Inc.)





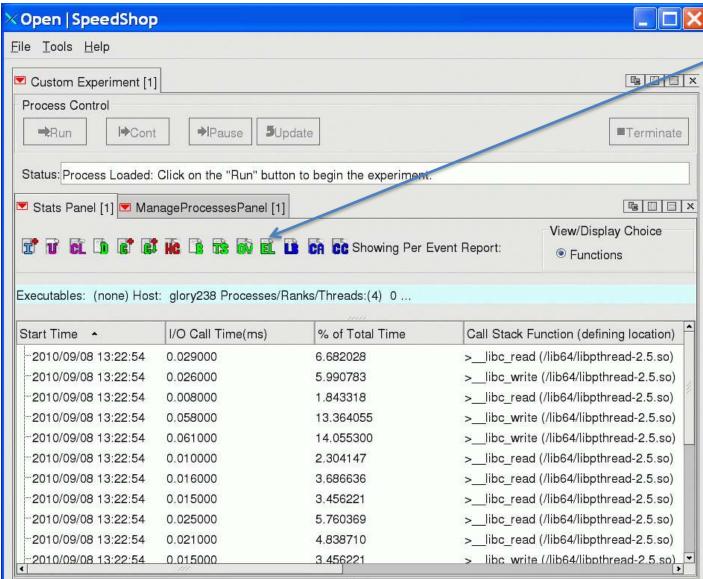
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define VARS_PER_CELL 15
/* Write a single restart file from many MPI processes */
int write restart (
 MPI_Comm comm
                             /// MPI communicator
 , int num_cells
                             /// number of cells on this process
 , double *cellv )
                             /// cell vector
                         // rank of this process within comm
 int rank;
 int size;
                         // size of comm
                         // for MPI Send, MPI Recv
 int tag;
                         // for serializing I/O
 int baton;
 FILE *f:
                        // file handle for restart file
 /* Procedure: Get MPI parameters */
 MPI Comm rank (comm, &rank);
 MPI_Comm_size (comm, &size);
 tag = 4747;
 if (rank == 0) {
  /* Rank 0 create a fresh restart file,
    * and start the serial I/O:
   * write cell data, then pass the baton to rank 1 */
   f = fopen ("restart.dat", "wb");
   fwrite (celly, num cells, VARS PER CELL * sizeof (double), f);
   fclose (f);
   MPI Send (&baton, 1, MPI INT, 1, tag, comm);
 } else {
```

```
/* Ranks 1 and higher wait for previous rank to complete I/O,
   * then append its cell data to the restart file,
   * then pass the baton to the next rank */
 MPI Recv (&baton, 1, MPI INT, rank - 1, tag, comm, MPI STATUS IGNORE);
  f = fopen ("restart.dat", "ab");
  fwrite (cellv, num_cells, VARS_PER_CELL * sizeof (double), f);
  fclose (f);
  if (rank < size - 1) {
   MPI_Send (&baton, 1, MPI_INT, rank + 1, tag, comm);
 /* All ranks have posted to the restart file; return to called */
return 0;
int main (int argc, char *argv[]) {
 MPI_Comm comm;
 int comm_rank;
 int comm_size;
 int num_cells;
 double *cellv;
 int i;
 MPI_Init (&argc, &argv);
 MPI_Comm_dup (MPI_COMM_WORLD, &comm);
 MPI_Comm_rank (comm, &comm_rank);
 MPI_Comm_size (comm, &comm_size);
  /**
  * Make the cells be distributed somewhat evenly across ranks
  */
  num_cells = 5000000 + 2000 * (comm_size / 2 - comm_rank);
  celly = (double *) malloc (num cells * VARS PER CELL * sizeof (double));
  for (i = 0; i < num_cells * VARS_PER_CELL; i++) {
   cellv[i] = comm_rank;
  write_restart (comm, num_cells, cellv);
  MPI_Finalize ();
return 0;
```

IOT O | SS Experiment of Serial I/O Example







SHOWS EVENT BY EVENT LIST:

Clicking on this gives each call to a I/O function being traced as shown.

Below is a graphical trace view of the same data showing serialization of fwrite() (THE RED BARS for each PE) with another tool.



Running I/O Experiments





Offline io/iop/iot experiment on sweep3d application

Convenience script basic syntax:

ossio[p][t] "executable" [default | <list of I/O func>]

- > Parameters
 - I/O Function list to sample(default is all)
 - creat, creat64, dup, dup2, lseek, lseek64, open, open64, pipe, pread, pread64, pwrite, pwrite64, read, readv, write, writev

Examples:

ossio "mpirun –np 256 sweep3d.mpi"

ossiop "mpirun –np 256 sweep3d.mpi" read,readv,write

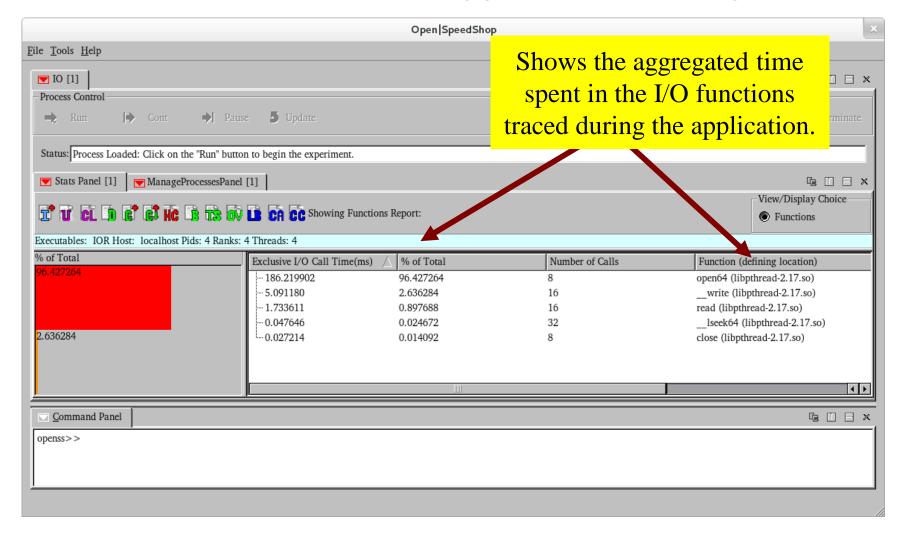
ossiot "mpirun –np 256 sweep3d.mpi" read,readv,write

I/O output via GUI





❖ I/O Default View for IOR application "io" experiment

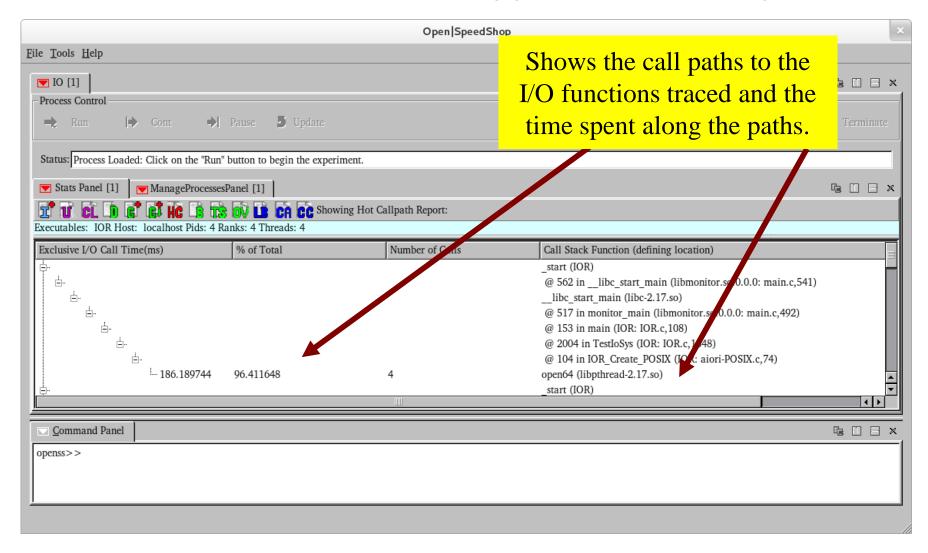


I/O output via GUI





❖ I/O Call Path View for IOR application "io" experiment



I/O output via CLI (equivalent of HC in GUI)





openss>>expview -vcalltrees,fullstack iot1

```
I/O Call Time(ms)
                   % of Total Time
                                     Number of Calls Call Stack Function (defining location)
                            start (sweep3d.mpi)
                              > @ 470 in libc start main (libmonitor.so.0.0.0: main.c,450)
                              >> libc start main (libc-2.10.2.so)
                               >>> @ 428 in monitor main (libmonitor.so.0.0.0: main.c,412)
                                >>>main (sweep3d.mpi)
                                 >>>> @ 58 in MAIN (sweep3d.mpi: driver.f,1)
                                 >>>> @ 25 in task_init_ (sweep3d.mpi: mpi_stuff.f,1)
                                   >>>>> gfortran ftell i2 sub (libgfortran.so.3.0.0)
                                    >>>>> gfortran ftell i2 sub (libgfortran.so.3.0.0)
                                        >>>>>> gfortran st read (libgfortran.so.3.0.0)
  17.902981000
                    96.220812461
                                            1 >>>>>>> libc read (libpthread-2.10.2.so)
```

Section Summary - I/O Tradeoffs





- Avoid writing to one file from all MPI tasks
 - If you need to, be sure to distinguish offsets for each PE at a stripe boundary, and use Buffered I/O
- If each process writes its own file, then the parallel file system attempts to load balance the Object Storage Targets (OSTs), taking advantage of the stripe characteristics
- ❖ Metadata server overhead can often create severe I/O problems
 - > Minimize number of files accessed per PE and minimize each PE doing operations like *seek*, *open*, *close*, *stat* that involve inode information
- ❖ I/O time is usually not measured, even in applications that keep some function profile
 - Open|SpeedShop can shed light on time spent in I/O using io, iot experiments

Hands-on Section 5: I/O Performance





- I/O experiments related application exercise
- ❖ I/O exercises can be found in these directories:
 - \$HOME/exercises/IOR
 - \$HOME/exercises/ser_par_io
- Consult README file in each of the directories for the instructions/guidance





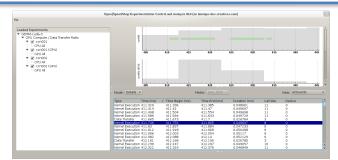
Open | SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 6 Analysis of Memory Usage







Memory Hierarchy





Memory Hierarchy

- > CPU registers and cache
- > System RAM
- > Online memory, such as disks, etc.
- > Offline memory not physically connected to system
- https://en.wikipedia.org/wiki/Memory hierarchy

What do we mean by memory?

- > Memory an application requires from the system RAM
- Memory allocated on the heap by system calls, such as malloc and friends

Need for Understanding Memory Usage





Memory Leaks

Is the application releasing memory back to the system?

Memory Footprint

- > How much memory is the application using?
- Finding the High Water Mark (HWM) of memory allocated
- Out Of Memory (OOM) potential
- > Swap and paging issues

Memory allocation patterns

- > Memory allocations longer than expected
- > Allocations that consume large amounts of heap space
- > Short lived allocations

Example Memory Heap Analysis Tools





MemP is a parallel heap profiling library

- > Requires mpi
- http://sourceforge.net/projects/memp

ValGrind provides two heap profilers.

- Massif is a heap profiler
 - http://valgrind.org/docs/manual/ms-manual.html
- DHAT is a dynamic heap analysis tool
 - http://valgrind.org/docs/manual/dh-manual.html

Dmalloc - Debug Malloc Library

- http://dmalloc.com/
- Google PerfTools heap analysis and leak detection.
 - https://github.com/gperftools/gperftools

O|SS Memory Experiment





Supports sequential, mpi and threaded applications.

- > No instrumentation needed in application.
- > Traces system calls via wrappers
 - malloc
 - calloc
 - realloc
 - free
 - memalign and posix_memalign

Provides metrics for

- > Timeline of events that set an new high-water mark.
- > List of event allocations (with calling context) to leaks.
- > Overview of all unique callpaths to traced memory calls that provides max and min allocation and count of calls on this path.

Example Usage

- > ossmem "./lulesh2.0"
- > ossmem "mpiexec mpt -np 64 ./sweep3d.mpi"

O|SS Memory Experiment CLI commands





expview -vunique

Show times, call counts per path, min,max bytes allocation, total allocation to all unique paths to memory calls that the mem collector saw

expview -vleaked

Show function view of allocations that were not released while the mem collector was active

expview -vtrace,leaked

> Will show a timeline of any allocation calls that were not released

expview -vfullstack,leaked

Display a full callpath to each unique leaked allocation

expview -v trace, highwater

- > Is a timeline of mem calls that set a new high-water
- The last entry is the allocation call that the set the high-water for the complete run
- ➤ Investigate the last calls in the timeline and look at allocations that have the largest allocation size (size1,size2,etc) if your application is consuming lots of system ram

O|SS Memory Experiment





Shows the last 8 allocation events that set the high water mark

openss>>expview -vtrace,highwater

```
Start Time(d:h:m:s) Event
                                         Ptr Return Value New Call Stack Function (defining location)
                           Size
                                  Size
                      lds
                           Arg1
                                  Arg2
                                         Arg
                                                         Highwater
*** trimmed all but the last 8 events of 61 ****
2016/11/10 09:56:50.824 11877:0
                                   2080
                                                 0x7760e0 19758988 >>>>> GI libc malloc (libc-
                                         0
2.18.so)
2016/11/10 09:56:50.826 11877:0 1728000
                                                 0x11783d0 21484908 >>> GI libc malloc (libc-
                                          0
2.18.so)
2016/11/10 09:56:50.827 11877:0 1728000
                                          0
                                                 0x131e1e0 23212908 >>>> GI libc malloc (libc-
2.18.so)
2016/11/10 09:56:50.827 11877:0 1728000
                                                 0x14c3ff0
                                                           24940908 >>>> GI libc malloc (libc-
                                          0
2.18.so)
2016/11/10 09:56:50.827 11877:0
                                   2080
                                          0
                                                 0x776a90 24942988 >>>>> GI libc malloc (libc-
2.18.so)
2016/11/10 09:56:50.919 11877:0 1728000
                                          0
                                                 0x1654030 25286604 >>> GI libc malloc (libc-
2.18.so)
2016/11/10 09:56:50.919 11877:0 1728000
                                                 0x17f9e40 27014604 >>>> GI libc malloc (libc-
                                          0
2.18.so)
2016/11/10 09:56:50.919 11877:0
                                   2080
                                          0
                                                  0xabc6a0 27016684 >>>>> GI libc malloc (libc-
2.18.so)
```

O|SS Memory Experiment





The next slide shows the default view of all unique memory calls seen while the mem collector was active. This is an overview of the memory activity. The default is display is aggregated across all processes and threads. Can view specific processes or threads.

For all memory calls the following are displayed:

- > The exclusive time and percent of exclusive time
- > The number of times this memory function was called.
- > The traced memory function name.

For allocation calls (e.g. malloc) the follow:

- > The max and min allocation size seen.
- > The number of times the that max or min was seen are displayed.
- > The total allocation size of all allocations.

O|SS Memory Experiment (Unique Calls)





openss>>expview -vunique

Exclusive (ms)	% of Total Time	Number of Calls		Min Requested Bytes	Max Request Count	Max Request Bytes	Total ed Bytes Requested	Function (defining location)
0.024847 0.002371	89.028629 8.495467	1546 5	1	192	6	4096	6316416	GIlibc_malloc (libc-2.18.so)GIlibc_free (libc-2.18.so)
0.000369	1.322154	1	1	40	1	40	40	realloc (libc-2.18.so)
0.000322	1.153750	3	1	368	1	368	1104	calloc (libc-2.18.so)

NOTE: Number of Calls means the number of unique paths to the memory function call.

To see the paths use the CLI command: expview –vunique, fullstack

O|SS Memory Experiment (Leaked Calls)





In this example the sequential OpenMP version of lulesh was run under ossmem.

The initial run detected 69 potential leaks of memory.

Examining the calltrees using the cli command "expview -vfullstack,leaked -mtot_bytes" revealed that allocations from the Domain::Domain constructor where not later released in the Domain::Domain destructor. After adding appropriate delete's in the destructor and rerunning ossmem, we observed a resolution of the leaks detected in the Domain class. The remaining leaks where minor and from system libraries.

Using the exprestore command to load in the initial database and the database from the second run, we can use the expcompare cli command to see the improvements. Below, database -x1 shows the initial run and -x2 shows the results from the run with the changes to address the leaks detected in the Domain class.

```
openss>>exprestore -f lulesh-mem-initial.openss
openss>>exprestore -f lulesh-mem-improved.openss
openss>>expcompare -vleaked -mtot_bytes -mcalls -x1 -x2
```

```
-x 2, Function (defining location)
 -x 1,
          -x 1, -x 2,
        Number Total
                         Number
 Total
          of
                           of
 Bytes
                 Bytes
Requested Calls Requested Calls
10599396
           69
                 3332
                               GI libc malloc (libc-2.17.so)
                              realloc (libc-2.17.so)
      72
            1
                   72
```

Summary and Conclusions





Benefits of Memory Heap Analysis

- > Detect leaks
- Inefficient use of system memory
- > Find potential OOM, paging, swapping conditions
- Determine memory footprint over lifetime of application run

Observations of Memory Analysis Tools

- > Less concerned with the time spent in memory calls
- Emphasis is placed on the relationship of allocation calls to free calls.
- > Can slow down and impact application while running

Hands-on Section 6: Memory Analysis





Memory experiment related application exercise

- More information provided at the tutorial
- Memory exercises can be found in these directories:
 - > \$HOME/exercises/matmul
 - > \$HOME/exercises/lulesh2.0.3
 - \$HOME/exercises/lulesh2.0.3-fixed
- ***** Look for the README file for instructions.





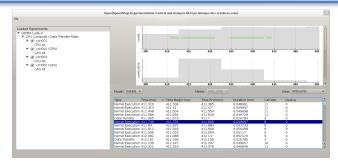
Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 7 Analysis of heterogeneous codes







Emergence of HPC Heterogeneous Processing





Heterogeneous computing refers to systems that use more than one kind of processor.

What led to increased heterogeneous processing in HPC?

- > Limits on ability to continue to scale processor frequencies
- Power consumption hitting realistic upper bound
- Programmability advances lead to more wide-spread, general usage of graphics processing unit (GPU).
- Advances in manycore, multi-core hardware technology (MIC)

Heterogeneous accelerator processing: (GPU, MIC)

- Data level parallelism (GPU)
 - Vector units, SIMD execution
 - Single instruction operates on multiple data items
- > Thread level parallelism (MIC)
 - Multithreading, multi-core, manycore

Overview: Most Notable Hardware Accelerators





GPU (Graphics Processing Unit)

- General-purpose computing on graphics processing units (GPGPU)
- Solve problems of type: Single-instruction, multiple thread (SIMT) model
- Vectors of data where each element of the vector can be treated independently
- > Offload model where data is transferred into/out-of the GPU
- Program using CUDA/OpenCL language or use directive based OpenACC

Intel MIC (Many Integrated Cores)

- Has a less specialized architecture than a GPU
- > Can execute parallel code written for:
 - Traditional programming models including POSIX threads, OpenMP
- Initially offload based (transfer data to and from co-processor)
- Now/future: programs to run natively

GPGPU Accelerator





GPU versus CPU comparison

Different goals produce different designs

- > GPU assumes work load is highly parallel
- > CPU must be good at everything, parallel or not

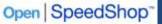
CPU: minimize latency experienced by 1 thread

- Big on-chip caches
- Sophisticated control logic

❖ GPU: maximize throughput of all threads

- # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
- Multi-threading can hide latency => skip the big caches
- Shared control logic across many threads

^{*}based on NVIDIA presentation



114

GPGPU Accelerator

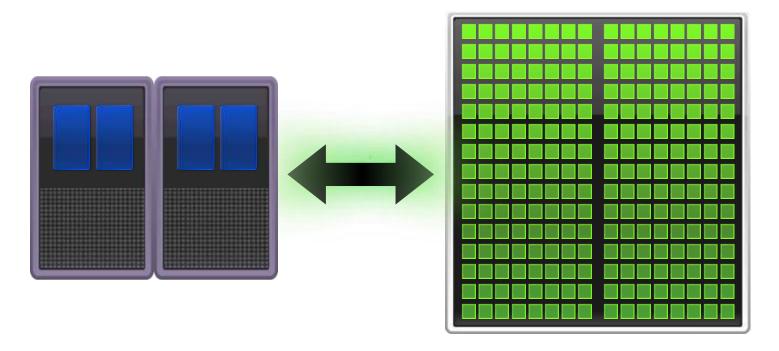




Mixing GPU and CPU usage in applications

Multicore CPU

Manycore GPU



Data must be transferred to/from the CPU to the GPU in order for the GPU to operate on it and return the new values.

^{*}NVIDIA image

Heterogeneous Programming





There are four main ways to use an accelerator

> Explicit programming:

 The programmer writes explicit instructions for the accelerator device to execute as well as instructions for transferring data to and from the device (e.g. CUDA-C for GPUs or OpenMP+Cilk Plus for Phis). This method requires to most effort and knowledge from programmers because algorithms must be ported and optimized on the accelerator device.

> Accelerator-specific pragmas/directives:

 Accelerator code is automatically generated from your serial code by a compiler (e.g. OpenACC, OpenMP 4.0). For many applications, adding a few lines of code (pragmas/directives) can result in good performance gains on the accelerator.

> Accelerator-enabled libraries:

• Only requires the use of the library, no explicit accelerator programming is necessary once the library has been written. The programmer effort is similar to using a non-accelerator enabled scientific library.

> Accelerator-aware applications:

 These software packages have been programed by other scientists/engineers/software developers to use accelerators and may require little or no programming for the end-user.

Credit: http://www.hpc.mcgill.ca/index.php/starthere/81-doc-pages/255-accelerator-overview

Programming for GPGPU





Prominent models for programming the GPGPU

Augment current languages to access GPU strengths

NVIDIA CUDA

- Scalable parallel programming model
- > Extensions to familiar C/C++ environment
- Heterogeneous serial-parallel computing
- Supports NVIDIA only

OpenCL (Open Computing Language)

- > Open source, royalty-free
- > Portable, can run on different types of devices
- Runs on AMD, Intel, and NVIDIA

OpenACC

- Provides directives (hint commands inserted into source)
- > Directives tell the compiler where to create acceleration (GPU) code without the user modifying or adapting the code.

Optimal Heterogeneous Execution





GPGPU considerations for best performance?

- How is the parallel scaling for the application overall?
- Can you balance the GPU and CPU workload?
 - Keep both the GPU and CPU busy for best performance
- Is it profitable to send a piece of work to the GPU?
 - What is the cost of the transfer of data to and from the GPU?
- How much work is there to be done inside the GPU?
 - > Will the work to be done fully populate and keep the GPU processors busy
 - > Are there opportunities to chain together operations so the data can stay in the GPU for multiple operations?
- Is there a vectorization opportunity?

Intel Phi considerations for best performance?

- Program should be heavily threaded
- Parallel scaling should be high with an OpenMP version

Accelerator Performance Monitoring





How can performance tools help optimize code?

- Is profitable to send a piece of work to the GPU?
 - > Can tell you this by measuring the costs:
 - Transferring data to and from the GPU
 - How much time is spent in the GPU versus the CPU
- Is there a vectorization opportunity?
 - Could measure the mathematical operations versus the vector operations occurring in the application
 - > Experiment with compiler optimization levels, re-measure operations and compare
- How is the parallel scaling for the application overall?
 - Use performance tool to get idea of real performance versus expected parallel speed-up
- Provide OpenMP programming model to source code insights
 - Use OpenMP performance analysis to map performance issues to source code

Open | SpeedShop accelerator support





What performance info does Open | SpeedShop provide?

- For GPGPU it reports information to help understand:
 - > Time spent in the GPU device
 - Cost and size of data transferred to/from the GPU
 - Balance of CPU versus GPU utilization
 - Transfer of data between the host and device memory versus the execution of computational kernels
 - Performance of the internal computational kernel code running on the GPU device
- Open|SpeedShop is able to monitor CUDA scientific libraries because it operates on application binaries.
- Support for CUDA based applications is provided by tracing actual CUDA events
- **❖** OpenACC support is conditional on the CUDA RT.

Open | SpeedShop accelerator support





What performance info does Open | SpeedShop provide?

❖ For Intel MIC (non-offload model):

- Reports the same range of performance information that it does for CPU based applications
- Open|SpeedShop will operate on MIC (co-processor KNC) similar to targeted platforms where the compute node processer is different than the front-end node processor
- > Only non-offload support is in our current plans
- ➤ A specific OpenMP profiling experiment (omptp) has been developed. Initial version is available now.
 - Will help to better support analysis of MIC based applications
 - OpenMP performance analysis key to understanding performance

121

CUDA GUI View: Default CUDA view

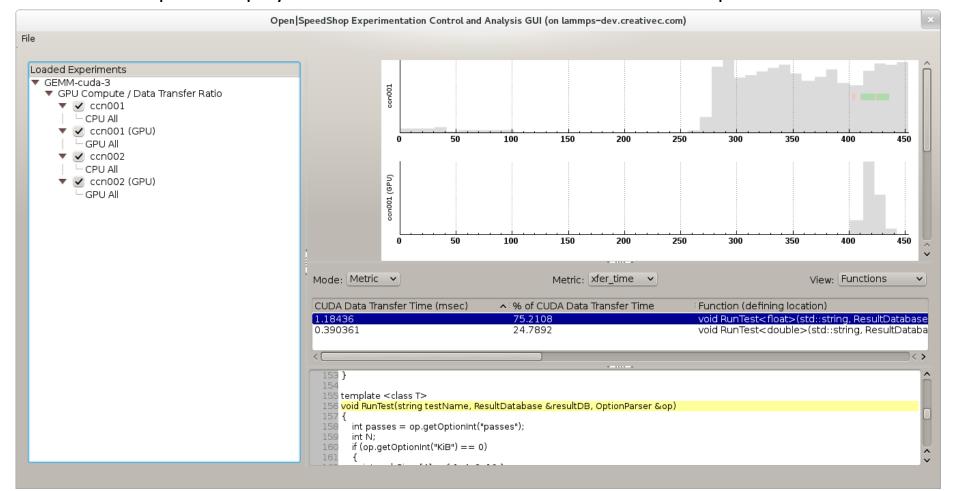




Note: The left pane shows the executable and the nodes it ran on. In future, will effect views. Internal GPU activity is shown in ccn0001 (GPU) graphic (shaded area)

Red lines indicate data transfers, Green lines indication GPU kernel executions

Source panel displays source for metrics clicked on in the Metric pane.



CUDA GUI View: Kernel Trace, 2nd GPU

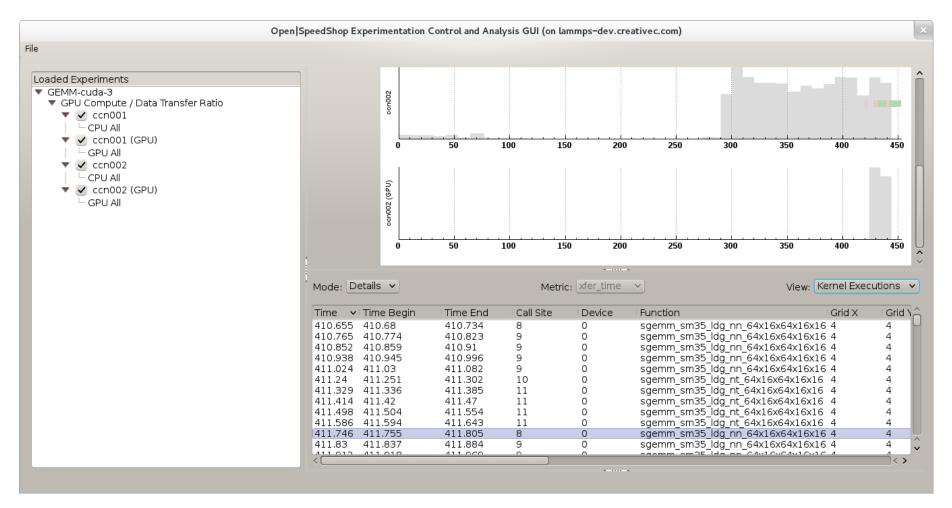




Note: The chronological list of kernel executions with details is in bottom pane.

Internal GPU activity is shown in ccn0002 (GPU) graphic (shaded area)

Red lines indicate data transfers, Green lines indication GPU kernel executions



CUDA GUI View: Transfers Trace, Zoom

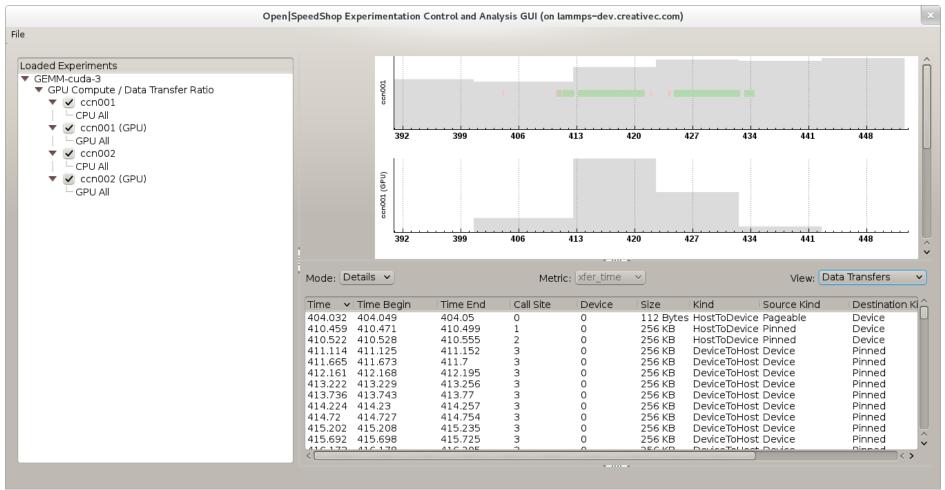




Note: The chronological list of data transfers with details is in bottom pane.

Internal GPU activity is shown in ccn0001 (GPU) graphic (shaded area)

Red lines indicate data transfers, Green lines indication GPU kernel executions



CUDA GUI View: All Events Trace



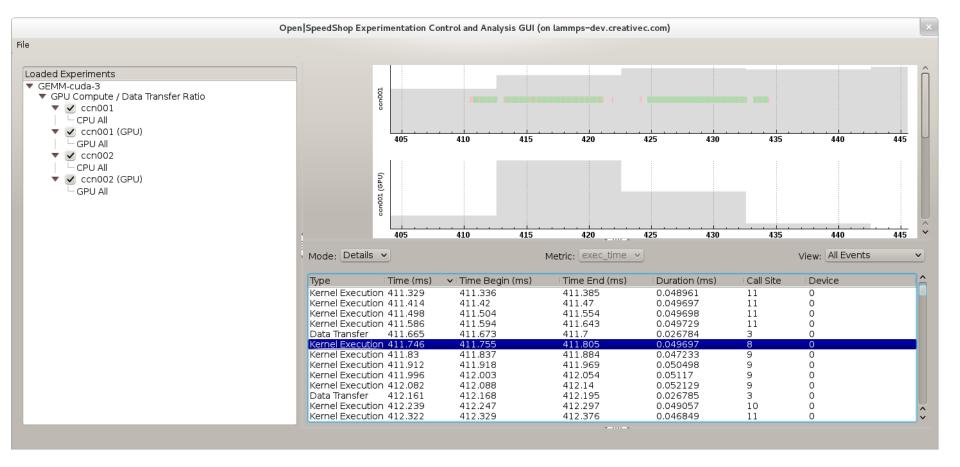


Note: The chronological list of data transfers and kernel executions in bottom pane.

Duration of kernel execution and data transfer available.

Internal GPU activity is shown in ccn0001 (GPU) graphic (shaded area)

Red lines indicate data transfers, Green lines indication GPU kernel executions



Open | SpeedShop CUDA CLI Views





```
openss>>expview [-vExec]
Exclusive
            % of Exclusive Function (defining location)
            Total
Time (ms)
                    Count
     Exclusive
        Time
                         300 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
14.810702 52.042113
                         300 void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
13.648369 47.957887
openss>>expview -vXfer
            % of Exclusive Function (defining location)
Exclusive
Time (ms)
            Total
                    Count
     Exclusive
        Time
1.774178 75.232917
                         69 void RunTest<float>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
                         69 void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)
0.584069 24.767083
openss>>expview -v trace, Xfer
                                                       Kind Call Stack Function (defining location)
 Start Time (d:h:m:s)
                          Exclusive
                                       % of
                                                Size
                         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)

Open | SpeedShop CUDA CLI Views





openss>>expview -v trace,Exec

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

2.753529 9.675400

> @ 137 in RunBenchmark(ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,122)

60 >> @ 231 in void RunTest<double>(std::string, ResultDatabase&, OptionParser&) (GEMM: GEMM.cpp,19)

Open | SpeedShop CUDA CLI Views





```
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)
```

_					
Time CP	U All	GPU All	<cpu< th=""><th> GPU></th><th>l</th></cpu<>	GPU>	l
(ms)	0757	•	***		
0 1586 11 533		0	*		
22 520		0	*		
33 541		0	*		
44 378		ŏ	199		
55 279		ŏ			
66 503		ŏ	*		
77 328		Ö	(C)		
88 224		ŏ			
99 162		ŏ	1		
110 67		ŏ			
	5549	ŏ	1		
	5052	ŏ	i I		
	4162	ŏ			
	3953	ŏ	1		
	6363	ŏ			
	5874	ŏ	1		
	2306	ŏ	1		
	4074	Ö			
	6671	ŏ			
	6696	Ö	i l		
	6431	Ö			
	3576	0	1		
	0303	Ō	i l		
264 493		0	*		
275 249		0	*****		
286 533	66059	0	*******		
297 755	79534	0	***********		
308 799	20340	0	*******		
319 766	04975	0	**********		
330 773	56196	0	***********		
341 788	01255	0	************		
352 683	18322	0	**********		
363 669	37166	0	**********		I .
374 694	01858	0	**********		İ
385 732		0	**********		İ
396 713	65211	544298	***********		ĺ
407 702		3554730	**********	****	į.
		10504920	**********	******	
429 828	53194	11857290	*********	*******	
440 687	40879	5299162	**********	******	Ì
451 206	65073	0	****		i

242	203576	0	I	
253	730303	0	I	l
264	4937670	0	*	1
275	24977312	0	*****	1
286	53366059	0	********	1
297	75579534	0	*********	1
308	79920340	0	*********	1
319	76604975	0	********	1
330	77356196	0	********	I
341	78801255	0	*********	1
352	68318322	0	**********	l
363	66937166	0	**********	l
374	69401858	0	**********	l I
385	73239976	0	**********	l I
396	71365211	544298	*********	l l
407	70238071	3554730	*********	****
418	70172897	10504920	******	*****
429	82853194	11857290	*******	******
440	68740879	5299162	*********	******
451	20665073	0	***	I I

Open | SpeedShop Accelerator support





What is the status of this Open | SpeedShop work?

Completed and being "hardened":

- > GPU related:
 - NVIDIA GPU Performance Data Collection
 - Recently integrated data collection to gather more information about the code being executing inside the GPU device
 - Packaging the data into the Open | SpeedShop database
 - Retrieving the performance data and formatting of text based views
 - Initial GUI based data views have been created in a new Qt4/Qt5 based GUI.
- MIC/Phi related
 - Basic initial support for Intel Phi (MIC) based applications.
 - First support was for KNC co-processor on MIC test beds at NERSC and at NASA on the maia Intel MIC platform.
 - Intel team has built and is using Open | SpeedShop for a KNL cluster.

On going work (NASA SBIR)

- More research and hardening of support for Intel PHI
- Analysis of performance data and more meaningful views for CUDA/GPU
- > Data reduction and clustering for scalability
- Continuation of GUI based views improvements.

Hands-on Section 7: GPU Performance





- GPU related application exercises
- GPU exercises can be found in these directories:
 - > \$HOME/exercises/cuda/shoc/bindir/bin/EP/CUDA
 - \$HOME/exercises/cuda/matrixMul
- Consult README for exercise instructions/guidance
 - > Run shoc benchmarks: GEMM and FFT





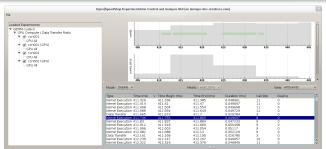
Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 8 Command Line Interface Usage











Command Line Interface Features

- "gdb" like tool for performance data creation and viewing
- > Same functional capabilities the graphical user interface (GUI)
 - Exception: GUI can focus on the source line corresponding to statistics
- > List metadata about your application and the OSS experiment
- Create experiments and run them
- > Launch the GUI from the CLI via the "opengui" command
- View performance data from a database file
 - openss –cli –f <database filename> to launch
 - expview key command with many options
 - list list many items such as source, object files, metrics to view
 - expcompare Compare ranks, threads, processes to each other, more...
 - cviewcluster Creates groups of like performing entities to outliers.
 - cview Output the columns of data representing groups created by cviewcluster.
 - cviewinfo Output what ranks, threads, etc., are in each cview group





Command Line Interface Features (additional)

- > Format the performance information view to csv
 - expview -F csv
- > Selectively only view, compare, analyze by rank, thread, process
 - -r <rank number> or rank list or rank ranges
 - -t <thread number> or thread list or thread ranges
 - -p process number> or process list or process ranges
- > Selectively only view, compare, analyze by specific metrics
 - -m <metric name> or list of metrics
 - Example metrics for sampling: percent, time
 - Example metrics for tracing: time, count, percent
- > Selectively only view by specific view type options
 - -v <view type> or list of view types
 - Example view types: functions, statements, loops, linked objects
 - Example metrics for tracing: time, count, percent





Command Line Interface Examples

- ❖ openss –cli –f <database file name>
- Commands to get started
 - > expstatus
 - Gives the metadata information about the experiment
 - > expview
 - Displays the default view for the experiment
 - Use expview <experiment type>nn to see only nn lines of output
 - expview pcsamp20 shows only the top 20 time taking functions
 - -v functions: displays data based on function level granularity
 - -v statements: displays data based on statement level granularity
 - -v linkedobjects: displays data based on library level granularity
 - -v loops: displays data based on loop level granularity
 - -v calltrees: displays call paths combining like paths
 - -v calltrees, fullstack : displays all unique call paths individually
 - -m loadbalance : displays the min, max, average values across ranks, ...





Command Line Interface Examples

- openss -cli -f <database file name>
- Commands to get started
 - expview (continued from previous page)
 - -v trace : for tracing experiments, display chronological list of events
 - -m <metric> : only display the metric(s) provided via the -m option
 - Where metric can be: time, percent, a hardware counter, (see list -v metrics)
 - -r <rank or rank list> : only display data for that rank or ranks
 - -t < thread id or list of thread ids> : only display data for that thread (s)
 - -h < host id or list of host ids> : only display data for that host or hosts
 - -F csv : display performance data in a comma separated list
 - expcompare : compare data within the same experiment
 - -r 1 -r 2 -m time : compare rank 1 to rank 2 for metric equal time
 - -h host1 -h host2 : compare host 1 to host 2 for the default metric





Command Line Interface Examples

- ❖ openss –cli –f <database file name>
- Commands to get started
 - > list
 - -v metrics: display the data types (metric) that can be displayed via –m
 - -v src : display source files associated with experiment
 - -v obj : display object files associated with experiment
 - -v ranks : display ranks associated with experiment
 - -v hosts : display machines associated with experiment
 - -v exp: display the experiment numbers that are currently loaded
 - -v savedviews: display the commands that are cached in the database

Viewing hwcsamp data in CLI





openss -cli -f smg2000-hwcsamp-1.openss

View the default report for this hwcsamp experiment

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

openss>>**expview**

Exclusive CPU time	% of CPU Time	PAPI_TOT_CYC	PAPI_FP_OPS Function (defining location)
in seconds.			
3.920000000 smg_residual.c,152)	44.697833523	11772604888	1198486900 hypre_SMGResidual (smg2000:
2.510000000 cyclic_reduction.c,757	28.620296465)	7478131309	812850606 hypre_CyclicReduction (smg2000:
0.310000000	3.534777651	915610917	48863259 opal_progress (libopen-pal.so.0.0.0)
0.300000000 semi_restrict.c,125)	3.420752566	910260309	100529525 hypre_SemiRestrict (smg2000:
0.29000000 (libmpi.so.0.0.2)	3.306727480	874155835	48509938 mca_btl_sm_component_progress

Viewing hwcsamp data in CLI





View the linked object (library) view for this Hardware Counter Sampling experiment

openss>>expview -v linkedobjects

Exclusive CPU time	% of CPU Time	PAPI_TOT_CYC	PAPI_FP_OPS LinkedObject
in seconds.			
7.710000000	87.315968290	22748513124	2396367480 smg2000
0.610000000	6.908267271	1789631493	126423208 libmpi.so.0.0.2
0.310000000	3.510758777	915610917	48863259 libopen-pal.so.0.0.0
0.200000000	2.265005663	521249939	46127342 libc-2.10.2.so
8.830000000	100.000000000	25975005473	2617781289 Report Summary
openss>>			





View the default I/O report for this I/O experiment

```
openss>> openss -cli -f sweep3d.mpi-iot.openss
```

```
openss>>[openss]: The restored experiment identifier is: -x 1
openss>>expview
I/O Call % of Number Function (defining location)
Time(ms) Total of
Time Calls
1.241909 90.077151 36 __write (libpthread-2.17.so)
0.076653 5.559734 2 close (libpthread-2.17.so)
0.035452 2.571376 2 read (libpthread-2.17.so)
```

View the default trace (chronological list of I/O functions calls) for this I/O experiment

2 open64 (libpthread-2.17.so)

openss>>expview -v trace

0.024703 1.791738

```
Start Time I/O Call % of Function
                                                          File/Path Name
                                                                              Event Call Stack Function (defining location)
            Time(ms) Total Dependent
                                                                   Identifier(s)
                        Return
                   Time
                       Value
2014/08/17 09:25:44.368 0.012356 0.896196
                                            13 input 0:140166697882752 >>>>>>pen64 (libpthread-2.17.so)
2014/08/17 09:25:44.368 0.027694 2.008679
                                                input 0:140166697882752 >>>>>>>read (libpthread-2.17.so)
2014/08/17 09:25:44.377 0.053832 3.904500
                                               input 0:140166697882752 >>>>>>close (libpthread-2.17.so)
2014/08/17 09:25:44.378 0.012347 0.895543
                                            13 input 0:140166697882752 >>>>>>pen64 (libpthread-2.17.so)
2014/08/17 09:25:44.378 0.007758 0.562697
                                            53 input 0:140166697882752 >>>>>>read (libpthread-2.17.so)
2014/08/17 09:25:44.378 0.022821 1.655235
                                             0 input 0:140166697882752 >>>>>>close (libpthread-2.17.so)
                                            62 /dev/pts/1 0:140166697882752 >>>>> write (libpthread-2.17.so)
2014/08/17 09:25:44.378 0.037219 2.699539
```





View the list of metrics (types of performance information) for this I/O experiment

openss>>list -v metrics

iot::average

iot::count

iot::exclusive_details

iot::exclusive_times

iot::inclusive_details

iot::inclusive_times

iot::max

iot::min

iot::nsysarg

iot::pathname

iot::retval

iot::stddev

iot::syscallno

iot::threadAverage

iot::threadMax

iot::threadMin

iot::time





View in chronological trace order: start_time, time, the rank:thread event occurred in.

```
openss>>expview -m start_time,time,id -vtrace
```

```
Start Time(d:h:m:s) Exclusive
                                  Event Call Stack Function (defining location)
                     Identifier(s)
            I/O Call
            Time(ms)
2014/08/17 09:25:44.368  0.012356 0:140166697882752 >>>>>>pen64 (libpthread-2.17.so)
2014/08/17 09:25:44.368  0.027694 0:140166697882752 >>>>>>>read (libpthread-2.17.so)
2014/08/17 09:25:44.377 0.053832 0:140166697882752 >>>>>>close (libpthread-2.17.so)
014/08/17 09:25:44.378  0.012347 0:140166697882752 >>>>>>>pen64 (libpthread-2.17.so)
2014/08/17 09:25:44.378 0.007758 0:140166697882752 >>>>>>>read (libpthread-2.17.so)
2014/08/17 09:25:44.378 0.022821 0:140166697882752 >>>>>>close (libpthread-2.17.so)
2014/08/17 09:25:44.378  0.037219 0:140166697882752 >>>>>> write (libpthread-2.17.so)
2014/08/17 09:25:44.378  0.018545 0:140166697882752 >>>>>> write (libpthread-2.17.so)
2014/08/17 09:25:44.378  0.019837 0:140166697882752 >>>>>> write (libpthread-2.17.so)
2014/08/17 09:25:44.379 0.035047 0:140166697882752 >>>>>> write (libpthread-2.17.so)
```

141





View the load balance (max, min, and average values) across all ranks, threads, or processes.

```
openss>>expview -m loadbalance
Max I/O Rank Min I/O Rank Average Function (defining location)
  Call of
          Call of I/O Call
  Time Max
               Time Min
                           Time
Across
            Across
                       Across
Ranks(ms)
             Ranks(ms)
                           Ranks(ms)
           0 1.241909
1.241909
                        0 1.241909 write (libpthread-2.17.so)
0.076653
           0 0.076653
                        0 0.076653 close (libpthread-2.17.so)
0.035452
           0 0.035452
                        0 0.035452 read (libpthread-2.17.so)
0.024703
           0 0.024703
                        0 0.024703 open64 (libpthread-2.17.so)
# View data for only rank nn, in this case rank 0
openss>>expview -r 0
I/O Call
          % of Number Function (defining location)
Time(ms)
           Total
                   of
       Time Calls
1.241909 90.077151
                      36 write (libpthread-2.17.so)
```

2 close (libpthread-2.17.so)

2 read (libpthread-2.17.so)

2 open64 (libpthread-2.17.so)

0.076653 5.559734 0.035452 2.571376

0.024703 1.791738

142





View the top time taking call tree in this application run. iot1 indicates see only one callstack. iot<number> shows "number" of calltrees.

```
openss>>expview -v calltrees,fullstack iot1
I/O Call
         % of Number Call Stack Function (defining location)
Time(ms)
           Total
                  of
       Time Calls
              start (sweep3d.mpi)
              > @ 562 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)
              >>>0x4026a2
              >>>> @ 185 in MAIN (sweep3d.mpi: driver.f,1)
             >>>> @ 41 in inner auto (sweep3d.mpi: inner auto.f,2)
              >>>>> @ 128 in inner (sweep3d.mpi: inner.f,2)
              >>>>> gfortran st write done (libgfortran.so.3.0.0)
              >>>>>>>>0x30fb8db56f
              >>>>>>>>>
              >>>>>>>>>>
0.871600 63.218195 12 >>>>>>> write (libpthread-2.17.so)
```





Open | SpeedShop CLI output into csv form for spreadsheet use.

Create an experiment database

```
EXE=./a.out export OPENSS_HWCSAMP_EVENTS="PAPI_VEC_DP,FP_COMP_OPS_EXE:SSE_FP_PACKED_DOUBLE" osshwcsamp "mpiexec_mpt -np 1 $EXE "
```

Open the database file and use expview –F csv to create a csv file openss –cli –f a.out-hwcsamp.openss –expview -F csv > mydata.csv

Create csv file using a script method

```
echo "exprestore -f a.out-hwcsamp.openss" > ./cmd_file
echo "expview -F csv > mydata.csv " >> ./cmd_file
echo "exit" >> ./cmd_file
# Run openss utility to output CSV rows.
#
openss -batch < ./cmd_file
```

GUI export of data to csv form





Getting Open | SpeedShop output into csv form for spreadsheet use.

openss –f stream.x-hwcsamp-1.openss

Go to Stats Panel Menu

Select the "Report Export Data (csv)" option

In the Dialog Box provide a meaningful name such as stream.x-hwcsamp-1.csv

File will be saved to a file you specified above

Command Line Interface (CLI) Usage





Storing csv information into spreadsheet

- Open your spreadsheet.
- Either
 - > Select open from pulldown menu
 - import stream.x-hwcsamp-1.csv
 - Cut and paste the contents of stream.x-hwcsamp-1.csv into the spreadsheet
- ❖ Use "Data" operation and then "Text to columns",
- ❖ Select "comma" to separate the columns.
- Save the spreadsheet file

Command Line Interface (CLI) Usage





Plotting the performance info from the spreadsheet (Libre office instructions)

- Open your spreadsheet.
- Select Insert, then choose Chart
- Choose a chart type
- Choose data range choices (data series in columns)
- Choose data series (nothing to do here)
- Chart is created. Views can be changed.





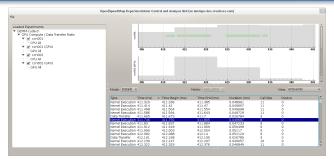
Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 9 Preferences





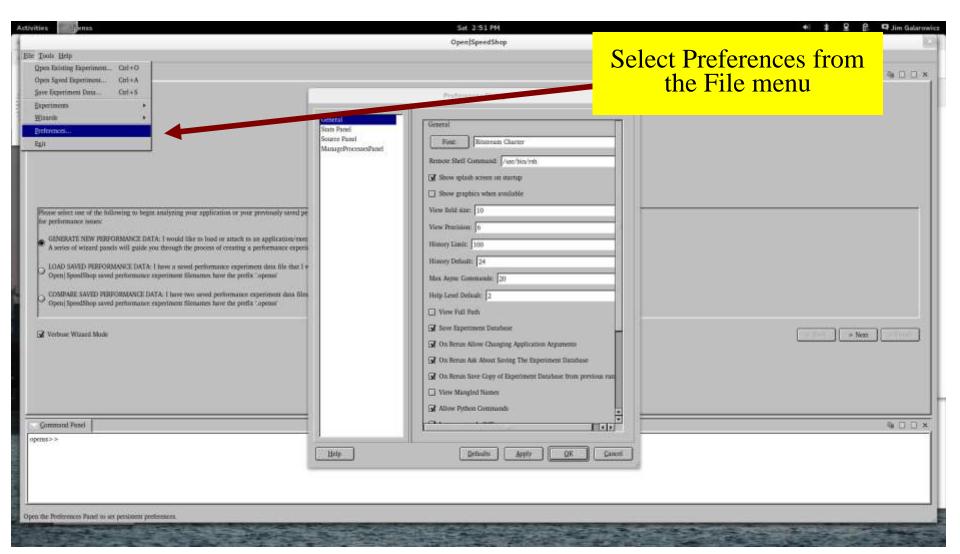


Changing Preferences in O|SS





Selecting File->Preferences to customize O | SS



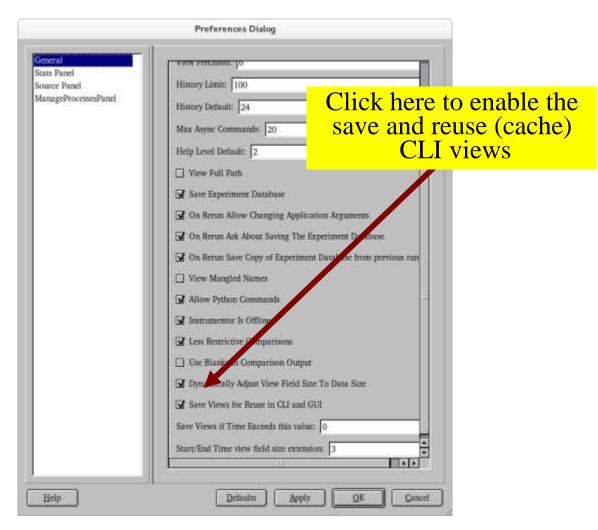
Changing Preferences in O|SS





Enabling the new Command Line Interface (CLI) save/reuse views feature. Looking for friendly user

evaluation.







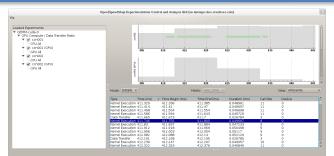
Open SpeedShop™



NASA Open | SpeedShop Tutorial Performance Analysis with Open | SpeedShop

Section 9 DIY & Conclusions







How to Take This Experience Home?





General questions should apply to ...

- > ... all systems
- ... all applications

Prerequisite

- Know what to expect from your application
- Know the basic architecture of your system

Ask the right questions

- > Start with simple overview questions
- > Dig deeper after that

❖ Pick the right tool for the task

- May need more than one tool
- Will depend on the question you are asking
- > May depend on what is supported on your system

If You Want to Give O|SS a Try?





Available on the these system architectures

- > AMD x86-64
- Intel x86, x86-64, MIC/Phi
- IBM PowerPC, PowerPC64. Power8
- > ARM: AArch64/A64 and AArch32/A32

Work with these operating system

- > Tested on Many Popular Linux Distributions
 - SLES, SUSE
 - RHEL, Fedora, CentOS
 - Debian, Ubuntu

Tested on some large scale platforms

- > IBM Blue Gene and Cray
- > GPU and Intel Phi support available
- > Available on many DOE/DOD systems in shared locations
- Ask your system administrator

How to Install Open | SpeedShop?





Most tools are complex pieces of software

- Low-level, platform specific pieces
- Complex dependencies
- > Need for multiple versions, e.g., based on MPIs and compilers
- Open|SpeedShop is no exception
 - In many cases even harder because of its transparency

Installation support

- > Three parts of the installation
 - Krell Root base packages
 - CBTF Component based tool framework
 - O|SS client itself
- > Install script
- Support for "spack" now available
 - https://github.com/scalability-llnl/spack

When in doubt, don't hesitate, ask us:

oss-contact@openspeedshop.org

Availability and Contact





- Current version: 2.3 has been released
- Open | SpeedShop Website
 - https://www.openspeedshop.org/
- Open | SpeedShop help and bug reporting
 - > Direct email: oss-contact@openspeedshop.org
 - Forum/Group: oss-questions@openspeedshop.org

❖ Feedback

- > Bug tracking available from website
- > Feel free to contact presenters directly
- > Support contracts and onsite training available

Getting Open | SpeedShop





Download options:

- Package with install script (install-tool)
- > Source for tool and base libraries

Project Wiki:

https://github.com/OpenSpeedShop/openspeedshop/wiki

Repositories access

https://github.com/OpenSpeedShop

Release Information

Release Tarball and Packages are accessible from www.openspeedshop.org

156

Open | SpeedShop Documentation





Build and Installation Instructions

- https://www.openspeedshop.org/documentation
 - Look for: Open | SpeedShop Version 2.3 Build/Install Guide

Open | SpeedShop User Guide Documentation

- https://www.openspeedshop.org/documentation
 - Look for Open | SpeedShop Version 2.3 Users Guide
- Man pages: OpenSpeedShop, osspcsamp, ossmpi,

•••

Quick start guide downloadable from web site

- https://www.openspeedshop.org
- > Click on "Download Quick Start Guide" button

Tutorial Summary





❖ Performance analysis critical on modern systems

- > Complex architectures vs. complex applications
- Need to break black box behavior at multiple levels
- Lots of performance left on the table by default

Performance tools can help

- Open | SpeedShop as one comprehensive option
- Scalability of tools is important
 - Performance problems often appear only at scale
 - We will see more and more online aggregation approaches
 - CBTF as one generic framework to implement such tools

Critical:

- > Asking the right questions
- Comparing answers with good baselines or intuition
- Starting at a high level and iteratively digging deeper

Questions vs. Experiments





Where do I spend my time?

- Flat profiles (pcsamp)
- Getting inclusive/exclusive timings with callstacks (usertime)
- Identifying hot callpaths (usertime + HP analysis)

How do I analyze cache performance?

- Measure memory performance using hardware counters (hwc)
- Compare to flat profiles (custom comparison)
- Compare multiple hardware counters (N x hwc, hwcsamp)

How to identify I/O problems?

- > Study time spent in I/O routines (io)
- Compare runs under different scenarios (custom comparisons)

How do I find parallel inefficiencies?

- > Study time spent in MPI routines (mpi)
- Look for load imbalance (LB view) and outliers (CA view)