SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Martin Schulz: LLNL

Mahesh Rajan: SNL

Donald Maghrak: Krell Institute

Jim Galarowicz: Krell Institute

Greg Scantlen: CreativeC

LLNL-PRES-503451

Why This Tutorial?

- **Performance Analysis is becoming more important**
  - Complex architectures and complex applications
  - Mapping applications onto architectures is hard
  - Today's applications only use a fraction of the machine
- **Performance analysis is more than just measuring time**
  - What are the critical sections in a code?
  - Is a part of the code running efficiently or not?
  - Is the code using the resources well (memory, TLB, I/O, …)?
  - Where is the greatest payoff for optimization?
- **Often hard to know where to start**
  - Which experiments to run first?
  - How to plan follow-on experiments?
  - What kind of problems can be explored?
  - How to interpret the data?

Tutorial Goals

- **Basic introduction into performance analysis**
  - Typical pitfalls wrt. performance
  - Wide range of types of performance tools and techniques
- **Provide basic guidance on …**
  - How to understand the performance of a code?
  - How to answer basic performance questions?
  - How to plan performance experiments?
- **Provide you with the ability to …**
  - Run these experiments on your own code
  - Provide starting point for performance optimizations
- **Practical Experience: Demos and hands-on Experience**
  - Introduction into Open|SpeedShop as one possible tool solution
  - Basic usage instructions and pointers to documentation
  - Lessons and strategies apply to any tool

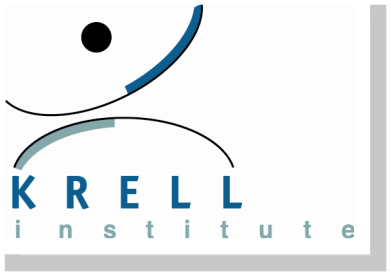
Open|SpeedShop Tool Set

- **Open Source Performance Analysis Tool Framework**
  - Most common performance analysis steps ***all in one tool***
  - Combines ***tracing*** and ***sampling*** techniques
  - ***Extensible*** by plugins for data collection and representation
  - 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*** and ***Blue Gene*** support

“Plan”/“Rules”

- **Staggered approach/agenda**
  - First two sessions: performance analysis basics
  - Third session: more specialized topics (I/O, memory, GPU)
  - Forth session: dedicated time for hands-on exercises
- **Let’s keep this interactive**
  - Feel free to ask questions as we go along
  - Ask if you would like to see anything specific in the demos
- **We are interested in feedback!**
  - What was clear / what didn’t make sense?
  - What scenarios are missing?
- **Updated slides available before SC**
  - http://www.openspeedshop.org/wp/category/tutorials
  - Then choose SC2015 Monday Nov 16 tutorial

Presenters



- **Martin Schulz: LLNL**
- **Mahesh Rajan: SNL**
- **Jim Galarowicz: Krell**
- **Donald Maghrak, Krell**
- **Greg Scantlen, CreativeC**
- **Larger team**
  - William Hachfeld and Dave Whitney: Krell
  - Jennifer Green, David Montoya, Mike Mason, David Shrader: LANL
  - Anthony Angelastos, SNL
  - Matt Legendre and Chris Chambreau: LLNL
  - Dyninst group (Bart Miller: UW & Jeff Hollingsworth: UMD)
  - Phil Roth: ORNL

Outline

- **Welcome**
- **Concepts in performance analysis**
- **Introduction into Tools and 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?**
- **<LUNCH>**
- **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**
  - On site cluster available
  - We will provide exercises and test codes

SC2015 Tutorial

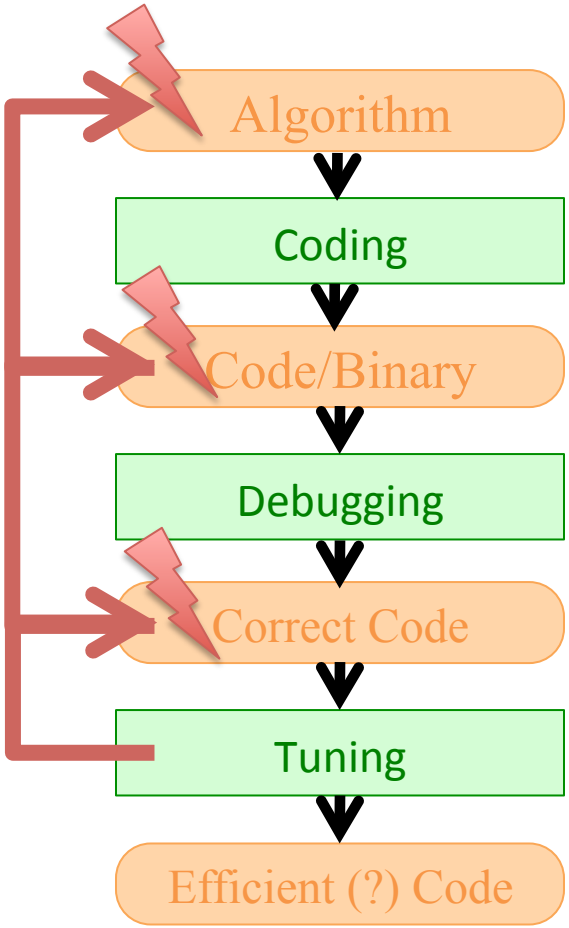
How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Section 1
Concepts in Performance Analysis

Typical Development Cycle

- **Performance tuning is an essential part of the development cycle**
  - Potential impact at every stage
    - Message patterns
    - Data structure layout
    - Algorithms
  - Should be done from early on in the life of a new HPC code
  - Ideally continuously and automatically
- **Typical use**
  - Measure performance and store data
  - Analyze data
  - Modify code and/or algorithm
  - Repeat measurements
  - ***Analyze differences***



A Case for Performance Tools

- **First line of defense**
  - Full execution timings (UNIX: "time" command)
  - Comparisons between input parameters
  - Keep and track historical trends
- **Disadvantages**
  - Measurements are coarse grain
  - Can't pin performance bottlenecks
- **Alternative: code integration of performance probes**
  - Hard to maintain
  - Requirements significant a priori knowledge
- **Performance tools**
  - Enable fine grain instrumentation
  - Show relation to source code
  - Work universally across applications

Performance Tools Overview

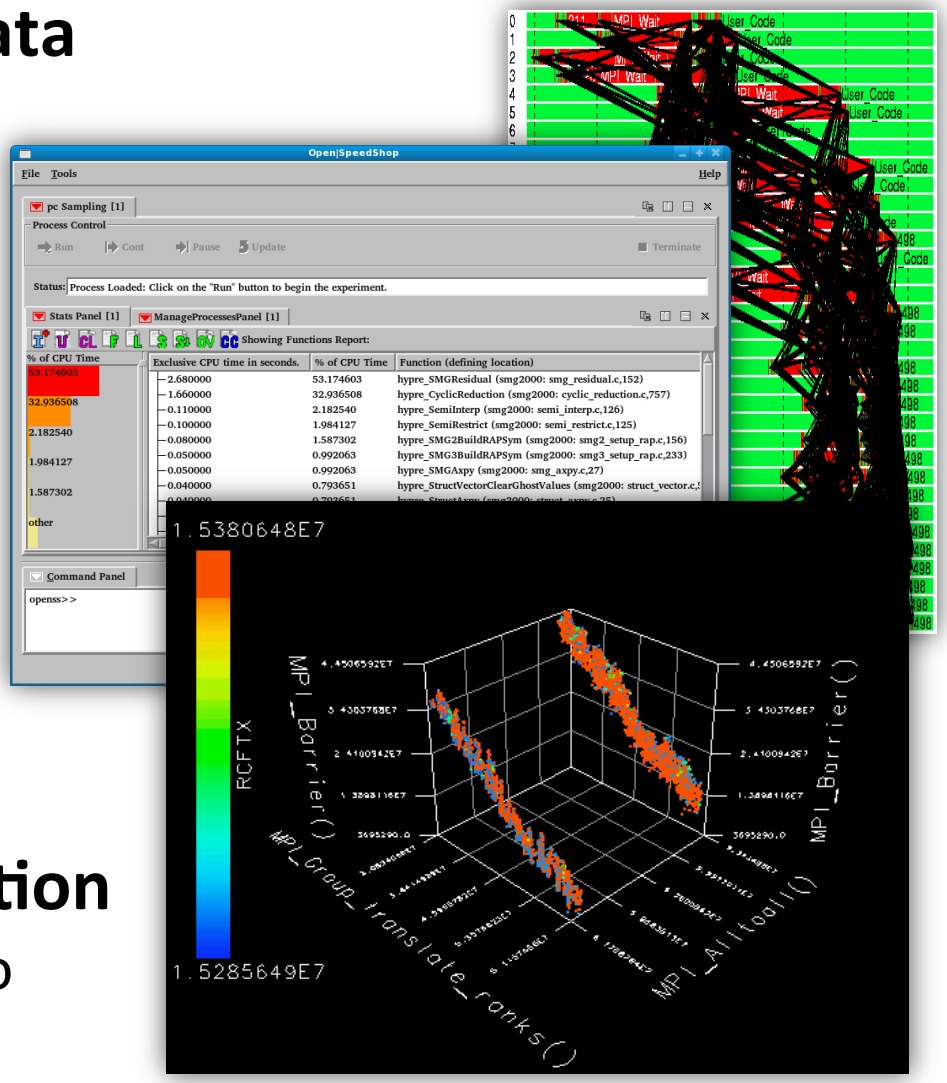
- **Basic OS tools**
  - time, gprof, strace
- **Hardware counters**
  - PAPI API & tool set
  - hwctime (AIX)
- **Sampling tools**
  - Typically unmodified binaries
  - Callstack analysis
  - HPCToolkit (Rice U.)
- **Profiling/direct measurements**
  - MPI or OpenMP profiles
  - mpiP (LLNL&ORNL)
  - ompP (LMU Munich)
- **Tracing tool kits**
  - Capture all MPI events
  - Present as timeline
  - Vampir (TU-Dresden)
  - Jumpshot (ANL)
- **Trace Analysis**
  - Profile and trace capture
  - Automatic (parallel) trace analysis
  - Kojak/Scalasca (JSC)
  - Paraver (BSC)
- **Integrated tool kits**
  - Typically profiling and tracing
  - Combined workflow
  - Typically GUI/some vis. support
  - Binary: Open|SpeedShop (Krell/TriLab)
  - Source: TAU (U. of Oregon)
- **Specialized tools/techniques**
  - Libra (LLNL)
    Load balance analysis
  - Boxfish (LLNL/Utah/Davis)
    3D visualization of torus networks
  - Rubik (LLNL)
    Node mapping on torus architectures
- **Vendor Tools**

How to Select a Tool?

- **A tool with the right features**
  - Must be easy to use
  - Provides performance analysis of the code at different levels: libraries, functions, loops, statements
- **A tool must match the application's workflow**
  - Requirements from instrumentation technique
    - Access to and knowledge about source code? Recompilation time?
    - Machine environments? Supported platforms?
  - Interactive and batch mode analysis options
  - Support iterative tuning with ability to compare key metrics across runs
- **Why We Picked/Developed Open|SpeedShop?**
  - Sampling and tracing in a single framework
  - Easy to use GUI & command line options for remote execution
    - Low learning curve for end users
  - Transparent instrumentation (preloading & binary)
    - No need to recompile application

Next Step: Interpret Data

- **Tools can collect lot's of data**
  - At varying granularity
  - At varying cost
  - At varying accuracy
- **Issue 1: Understand your tool and its limitations**
  - No tool can do everything (at least not well)
  - Choose the right tool for the right task
- **Issue 2: Ask the right question**
  - Need to know basic issues to look for to get started
  - Need to understand expected behavior



Issue 1: Tool Types

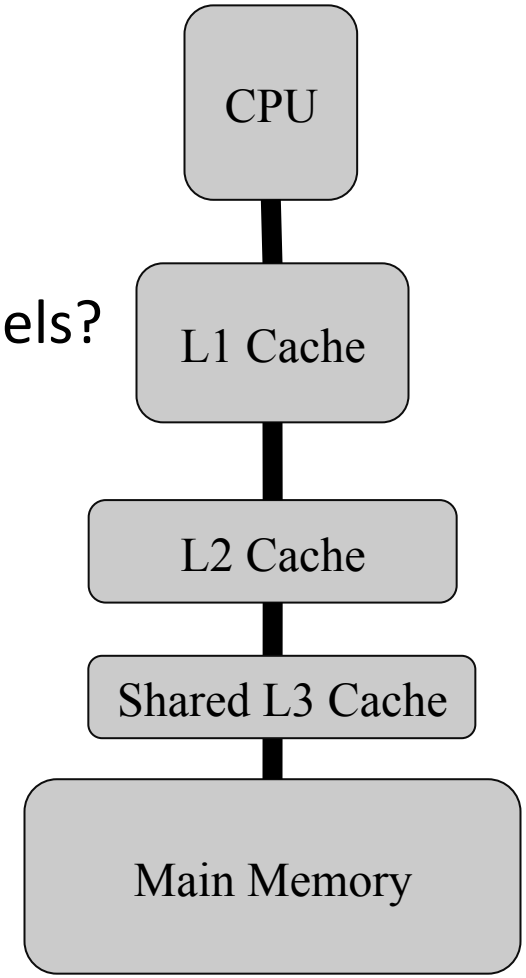
- **Data acquisition**
  - Event based data: triggered by explicit events
    - Direct correlation possible, but may come in bursts
  - Sampling based data: triggered by external events like timers
    - Even distribution, but requires statistical analysis
- **Instrumentation**
  - Source code instrumentation: exact, but invasive
  - Compiler instrumentation: requires source, but transparent
  - Binary instrumentation: can be transparent, but still costly
  - Link-level: transparent, less costly, but limited to APIs
  - Tradeoff: invasiveness vs. overhead vs. ability to correlate
  - Big question: granularity
- **Aggregation**
  - No aggregation: trace
  - Aggregation over time and space: simplified profile
  - Many shades of gray in between

Issue 2: Asking the Right Questions

- **Step 1: Find where the problem actually is**
  - Where is the code spending time?
    - Which code sections are even worth look at?
  - Where should it spend time?
    - Have a (mental) model of your application
- **Use overview experiments**
  - Identify bottlenecks for your application
    - Which resource in the system is holding you back?
  - Decide where to dig deeper
    - Important resource AND worth optimizing AND unexpected behavior
- **Pick the right tool or experiment in a tool**
  - Target the specific bottleneck
  - Decide on instrumentation approach
  - Decide on useful aggregation
  - Understand impact on code perturbation

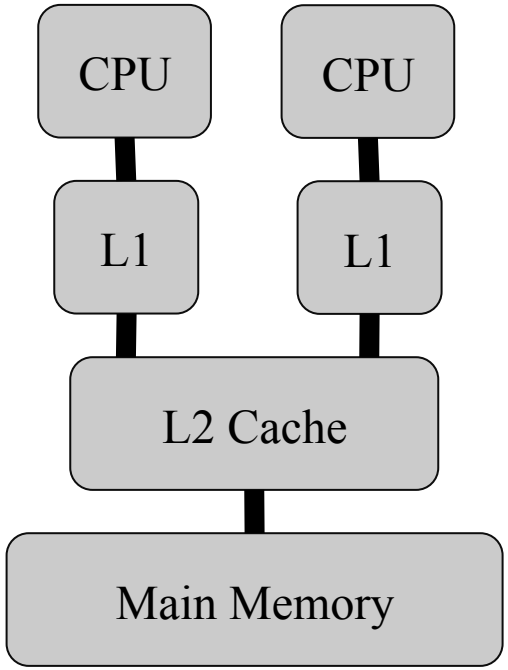
What to Look For: Sequential Runs

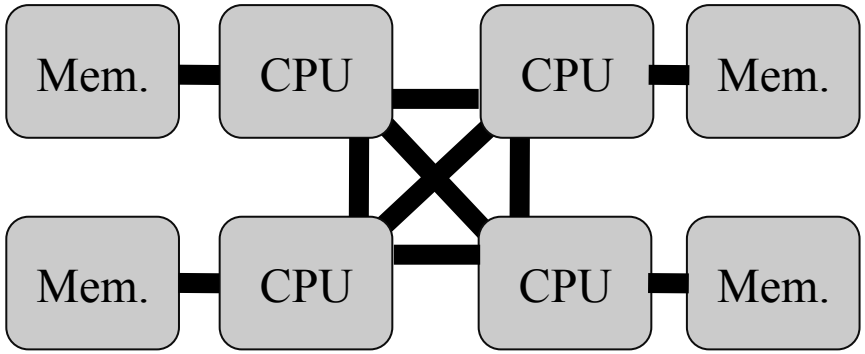
- **Step 1: Identify computational intensive parts**
  - Where am I spending my time?
    - Modules/Libraries
    - Loops
    - Statements
    - Functions
  - Is the time spent in the computational kernels?
  - Does this match my intuition?
- **Impact of memory hierarchy**
  - Do I have excessive cache misses?
  - How is my data locality?
  - Impact of TLB misses?
- **External resources**
  - Is my I/O efficient?
  - Time spent in system libraries?



What to Look For: Shared Memory

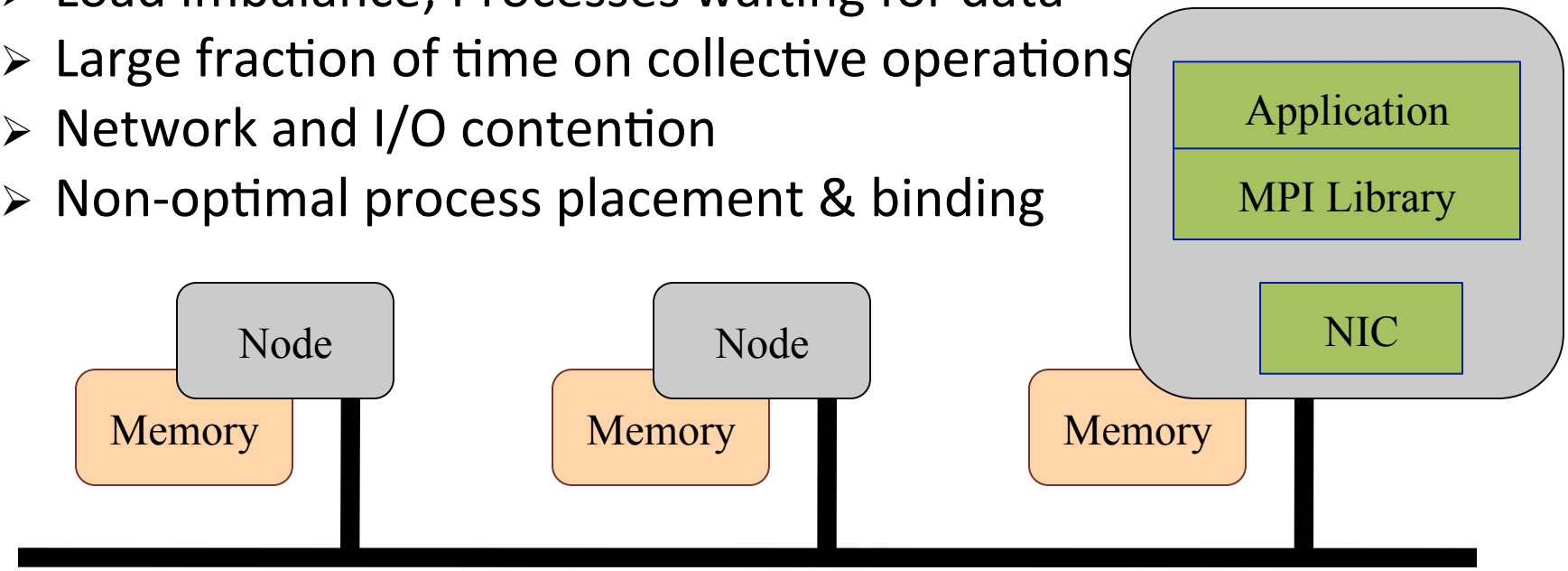
- **Shared memory model**
  - Single shared storage
  - Accessible from any CPU
- **Common programming models**
  - Explicit threads (e.g., POSIX threads)
  - OpenMP
- **Typical performance issues**
  - False cache sharing
  - Excessive Synchronization
  - Limited work per thread
  - Threading overhead
- **Complications: NUMA**
  - Memory locality critical
  - Thread:Memory assignments





What to Look For: Message Passing

- **Distributed Memory Model**
  - Sequential/shared memory nodes coupled by a network
  - Only local memory access
  - Data exchange using message passing (e.g., MPI)
- **Typical performance issues**
  - Load imbalance; Processes waiting for data
  - Large fraction of time on collective operations
  - Network and I/O contention
  - Non-optimal process placement & binding



What's Next

- **Overview of Open|SpeedShop**
  - Help to understand demos and hands-on exercises
- **Basic questions**
  - Where am I spending my time?
  - How to understand the context of this information?
- **Hardware/Resource utilization**
  - How to use hardware counters efficiently?
  - How to turn this information into actionable insight?
- **Next step beyond the computational core**
  - How well is my I/O doing?
  - How well am I utilizing memory?
  - How can I understand the performance on accelerators?

SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Section 2
Introduction into Tools and Open|SpeedShop

Open|SpeedShop Tool Set

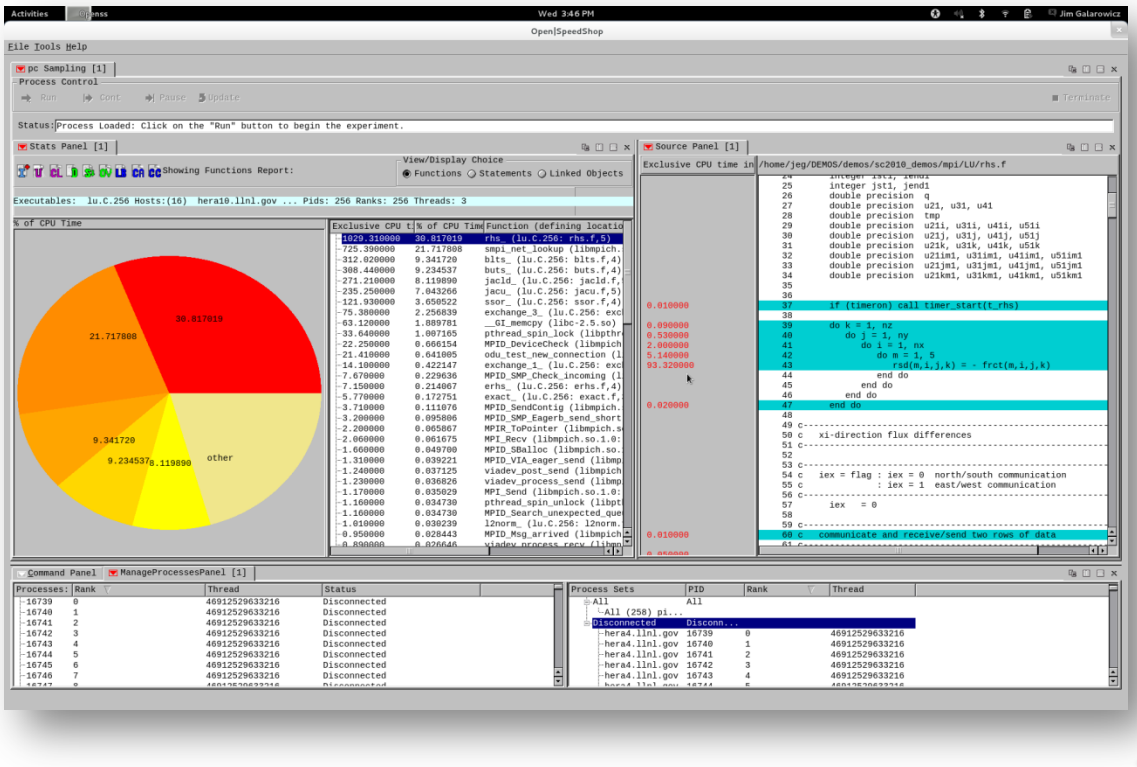
- **Open Source Performance Analysis Tool Framework**
  - Most common performance analysis steps ***all in one tool***
  - Combines ***tracing*** and ***sampling*** techniques
  - ***Extensible*** by plugins for data collection and representation
  - Gathers and displays several types of performance 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, Intel MIC, GPU*** support

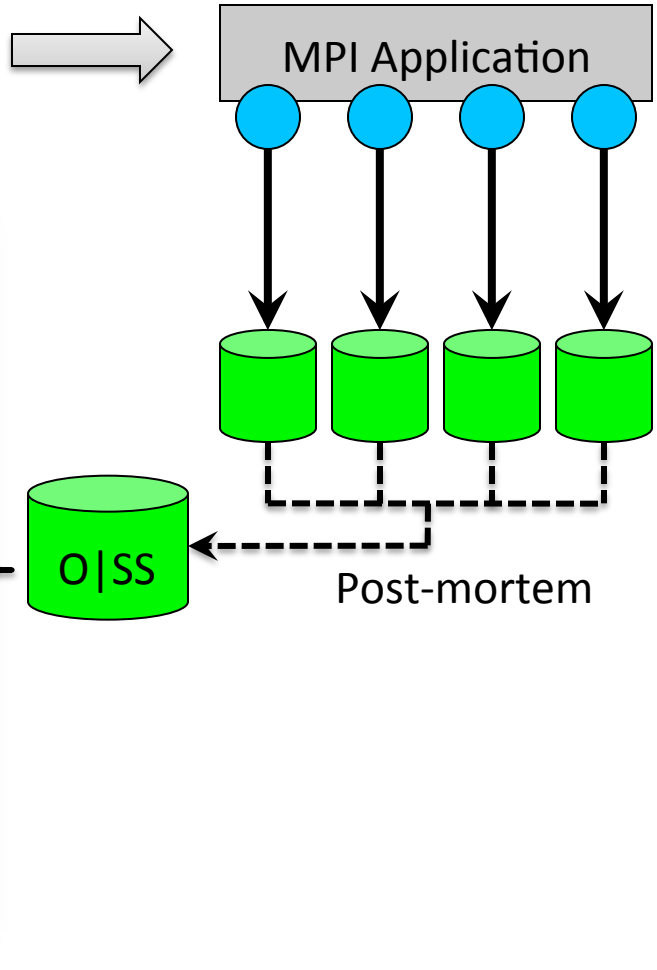
Classifying Open|SpeedShop

- **Offers both sampling and direct instrumentation**
  - Sampling: for overview and hardware counter experiments
  - Instrumentation for communication/memory events
    - Instrumentation at API level: MPI, pthreads, OpenMP, …
    - Some support for user instrumentation
- **Instrumentation**
  - At the link level
    - Well suited for API level instrumentation
    - Can still cause bursty overhead
  - Loop analysis based on binary instrumentation techniques
    - Executed post mortem for sampling experiments
    - Overhead outside the critical both to avoid perturbation
- **Aggregation**
  - By default: profile over time intervals
    - Full traces possible for some experiments (e.g., MPI) but costly
  - Aggregation over processes possible, but not default
    - Enables user to query per process/thread data

Open|SpeedShop Workflow

osspcsamp “srun –n4 –N1 smg2000 –n 65 65 65”





http://www.openspeedshop.org/

Alternative Interfaces

- **Scripting language**
  - Immediate command interface
  - O|SS interactive command line (CLI)
    - openss -cli

```
Experiment Commands
    expView
    expCompare
    expStatus

List Commands
    list –v exp
    list –v hosts
```

- **Python module**

```
import openss

my_filename=openss.FileList("myprog.a.out")
my_exptype=openss.ExpTypeList("pcsamp")
my_id=openss.expCreate(my_filename,my_exptype)

openss.expGo()

My_metric_list = openss.MetricList("exclusive")
my_viewtype = openss.ViewTypeList("pcsamp”)
result = openss.expView(my_id,my_viewtype,my_metric_list)
```

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:**
  - 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)**
  - Record invocation of all POSIX I/O events
  - Provides aggregate and individual timings
  - Store function arguments and return code for each call (iot)
- **MPI Tracing (mpi, mpit, mpiotf)**
  - Record invocation of all MPI routines
  - Provides aggregate and individual timings
  - Store function arguments and return code for each call (mpit)
  - Create Open Trace Format (OTF) output (mpiotf)
- **Floating Point Exception Tracing (fpe)**
  - Triggered by any FPE caused by the application
  - Helps pinpoint numerical problem areas

Additional Experiments in OSS/CBTF

- **POSIX thread tracing (pthreads)**
  - Record invocation of all POSIX thread events
  - Provides aggregate and individual rank, thread, or process timings
- **MPI Tracing (mpip)**
  - Record invocation of all MPI routines
  - Provides aggregate and individual rank, thread, or process timings
  - Lightweight MPI profiling because not tracking individual call details
- **Memory Tracing (mem)**
  - Record invocation of key memory related function call events
  - Provides aggregate and individual rank, thread, or process timings
- **Input/Output Tracing (iop)**
  - Record invocation of all POSIX I/O events
  - Provides aggregate and individual rank, thread, or process timings
  - Lightweight I/O profiling (iop)
- **CUDA NVIDIA GPU Event Tracing (cuda)**
  - Record CUDA events, provides timeline and event timings

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 store function arguments and return code for each call

How to Run a First Experiment in O|SS?

1. **Picking the experiment**
   - What do I want to measure?
   - *We will start with pcsamp to get a first overview*
2. **Launching the application**
   - How do I control my application under O|SS?
   - *Enclose how you normally run your application in quotes*
   - ***osspcsamp*** *"mpirun –np 256 smg2000 –n 65 65 65"*
3. **Storing the results**
   - *O|SS will create a database*
   - *Name: smg2000-pcsamp.openss*
4. **Exploring the gathered data**
   - How do I interpret the data?
   - *O|SS will print a default report*
   - *Open the GUI to analyze data in detail (run: "**openss**")*

Example Run with Output

❖ osspcsamp “mpirun –np 2 smg2000 –n 65 65 65” (1/2)

```
Bash> osspcsamp "mpirun -np 2 ./smg2000 -n 65 65 65"
[openss]: pcsamp experiment using the pcsamp experiment default sampling rate: "100".
[openss]: Using OPENSS_PREFIX installed in /opt/OSS-mrnet
[openss]: Setting up offline raw data directory in /tmp/jeg/offline-oss
[openss]: Running offline pcsamp experiment using the command:
"mpirun -np 2 /opt/OSS-mrnet/bin/ossrun "./smg2000 -n 65 65 65" pcsamp"

Running with these driver parameters:
 (nx, ny, nz)    = (65, 65, 65)
 …
        <SMG native output>
…
Final Relative Residual Norm = 1.774415e-07
[openss]: Converting raw data from /tmp/jeg/offline-oss into temp file X.0.openss

Processing raw data for smg2000
Processing processes and threads ...
Processing performance data ...
Processing functions and statements ...
```

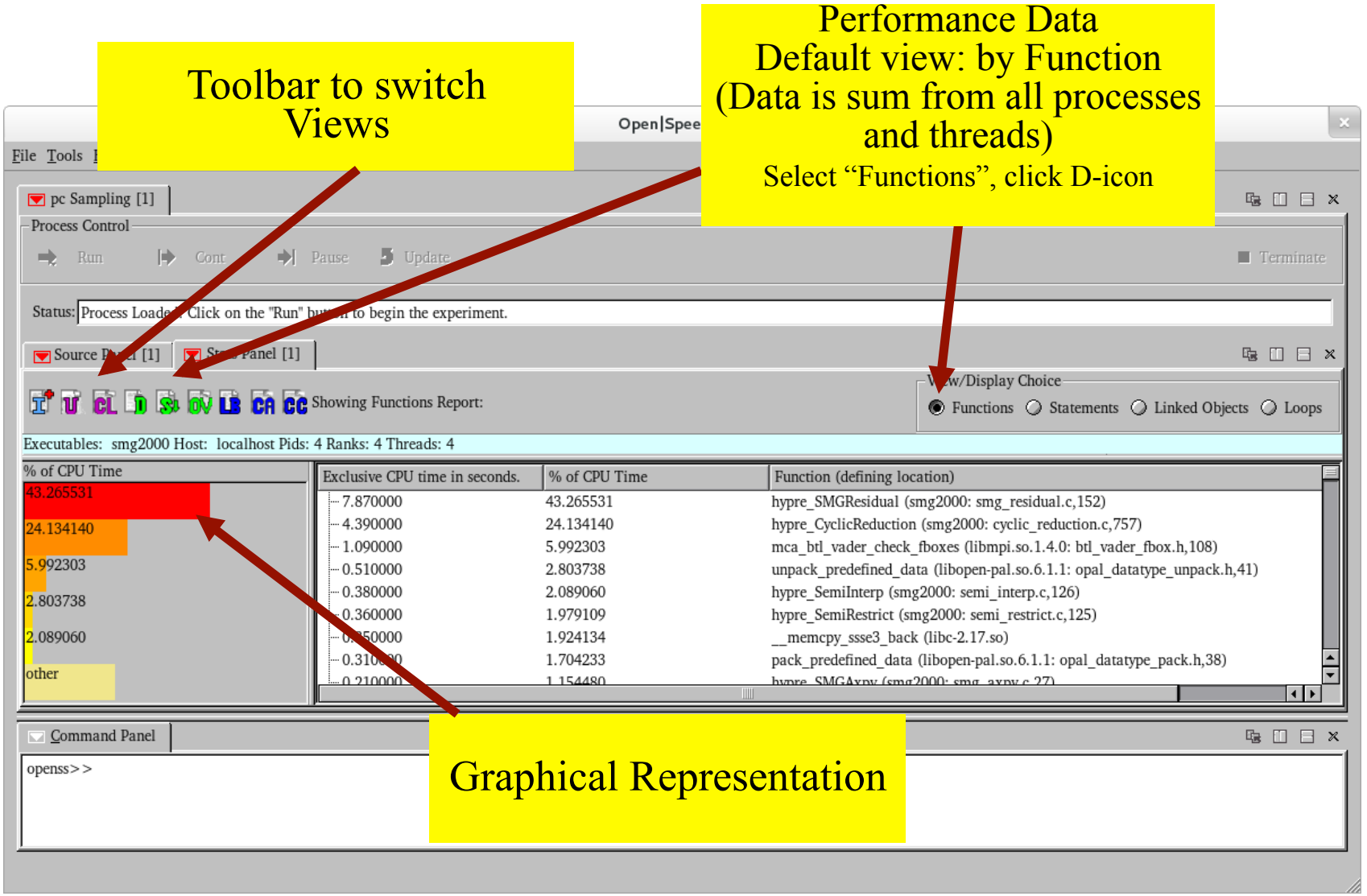
Example Run with Output

❖ osspcsamp "mpirun –np 2 smg2000 –n 65 65 65" (2/2)

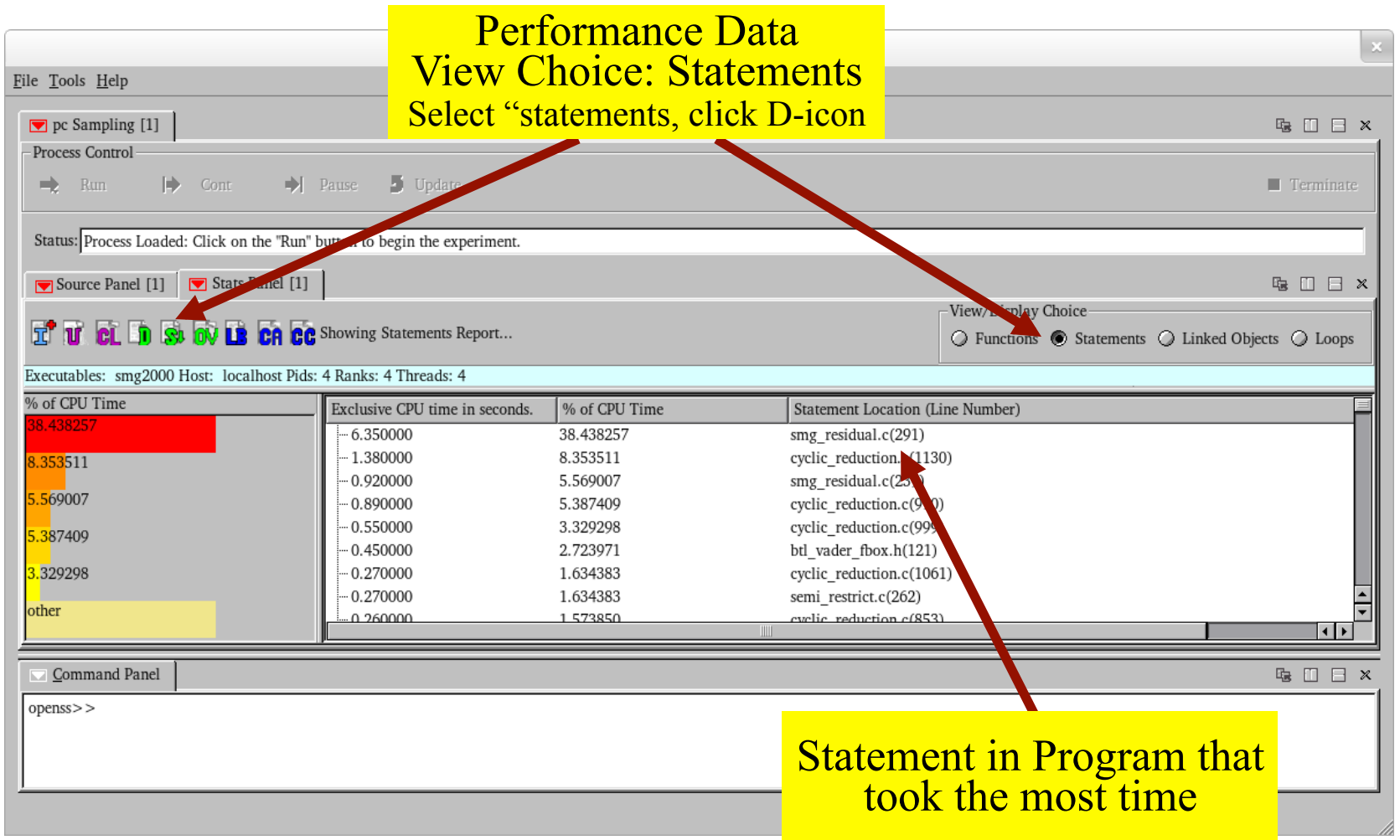
```
[openss]: Restoring and displaying default view for:
        /home/jeg/DEMOS/demos/mpi/openmpi-1.4.2/smg2000/test/smg2000-pcsamp.openss
[openss]: The restored experiment identifier is:  -x 1

 Exclusive CPU time      % of CPU Time  Function (defining location)
      in seconds.
      3.630000000       43.060498221  hypre_SMGResidual (smg2000: smg_residual.c,152)
      2.860000000       33.926453144  hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)
      0.280000000        3.321470937  hypre_SemiRestrict (smg2000: semi_restrict.c,125)
      0.210000000        2.491103203  hypre_SemiInterp (smg2000: semi_interp.c,126)
      0.150000000        1.779359431  opal_progress (libopen-pal.so.0.0.0)
      0.100000000        1.186239620 mca_btl_sm_component_progress (libmpi.so.0.0.2)
      0.090000000        1.067615658  hypre_SMGAxpy (smg2000: smg_axpy.c,27)
      0.080000000        0.948991696  ompi_generic_simple_pack (libmpi.so.0.0.2)
      0.070000000        0.830367734  __GI_memcpy (libc-2.10.2.so)
      0.070000000        0.830367734 hypre_StructVectorSetConstantValues (smg2000:
struct_vector.c,537)
      0.060000000        0.711743772  hypre_SMG3BuildRAPSym (smg2000: smg3_setup_rap.c,233)
```

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

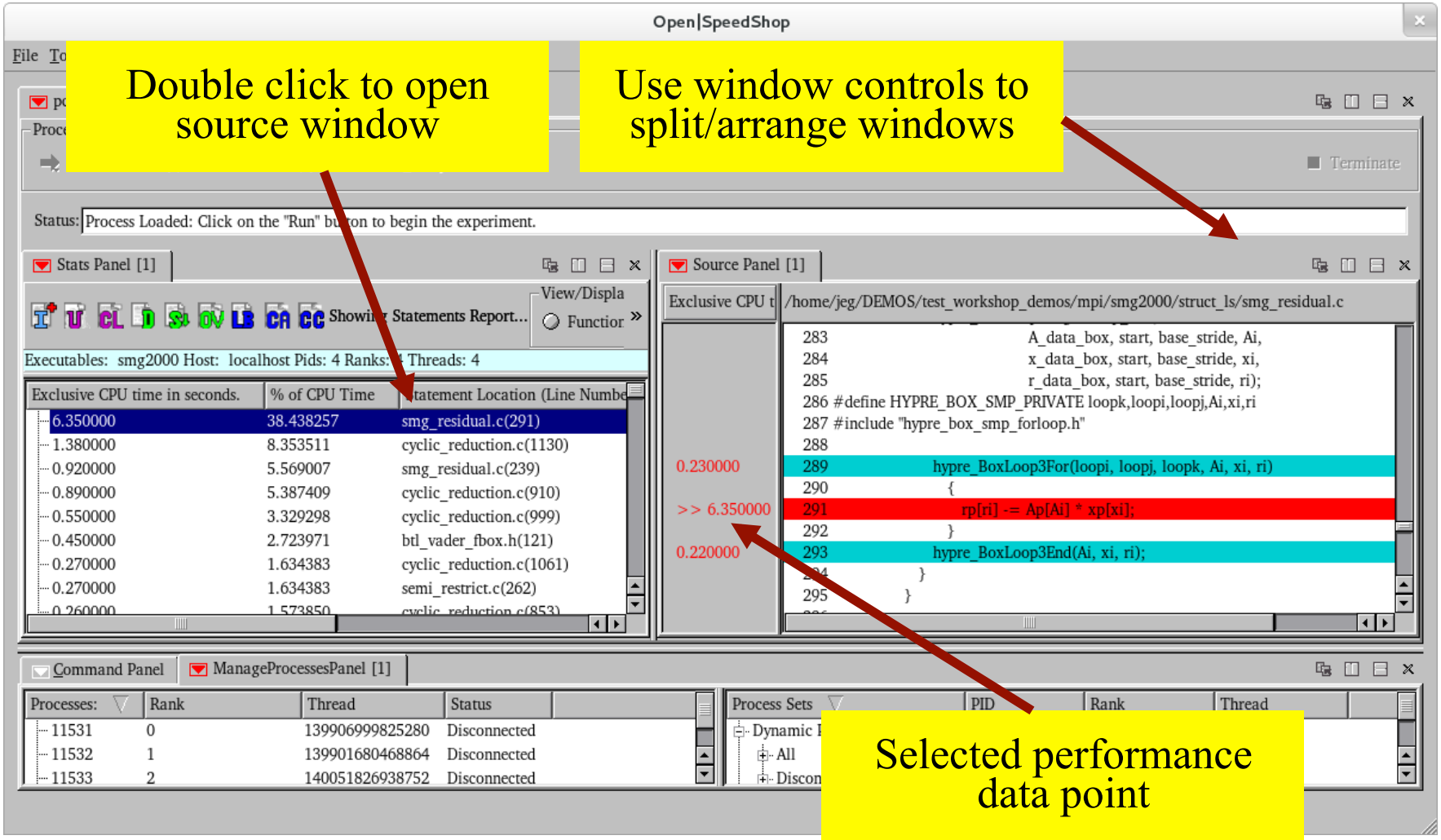
Default Output Report View



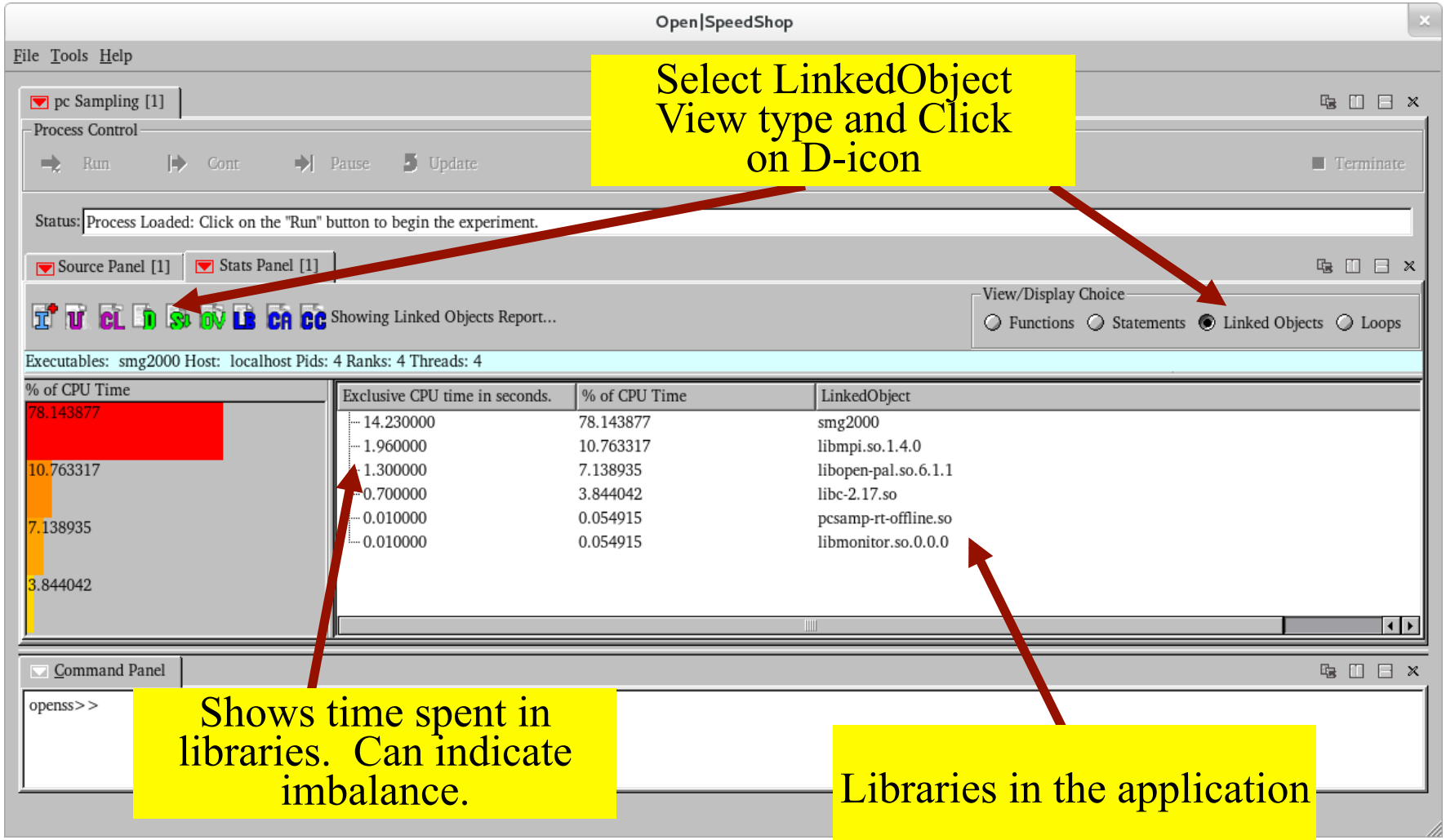
Statement Report Output View



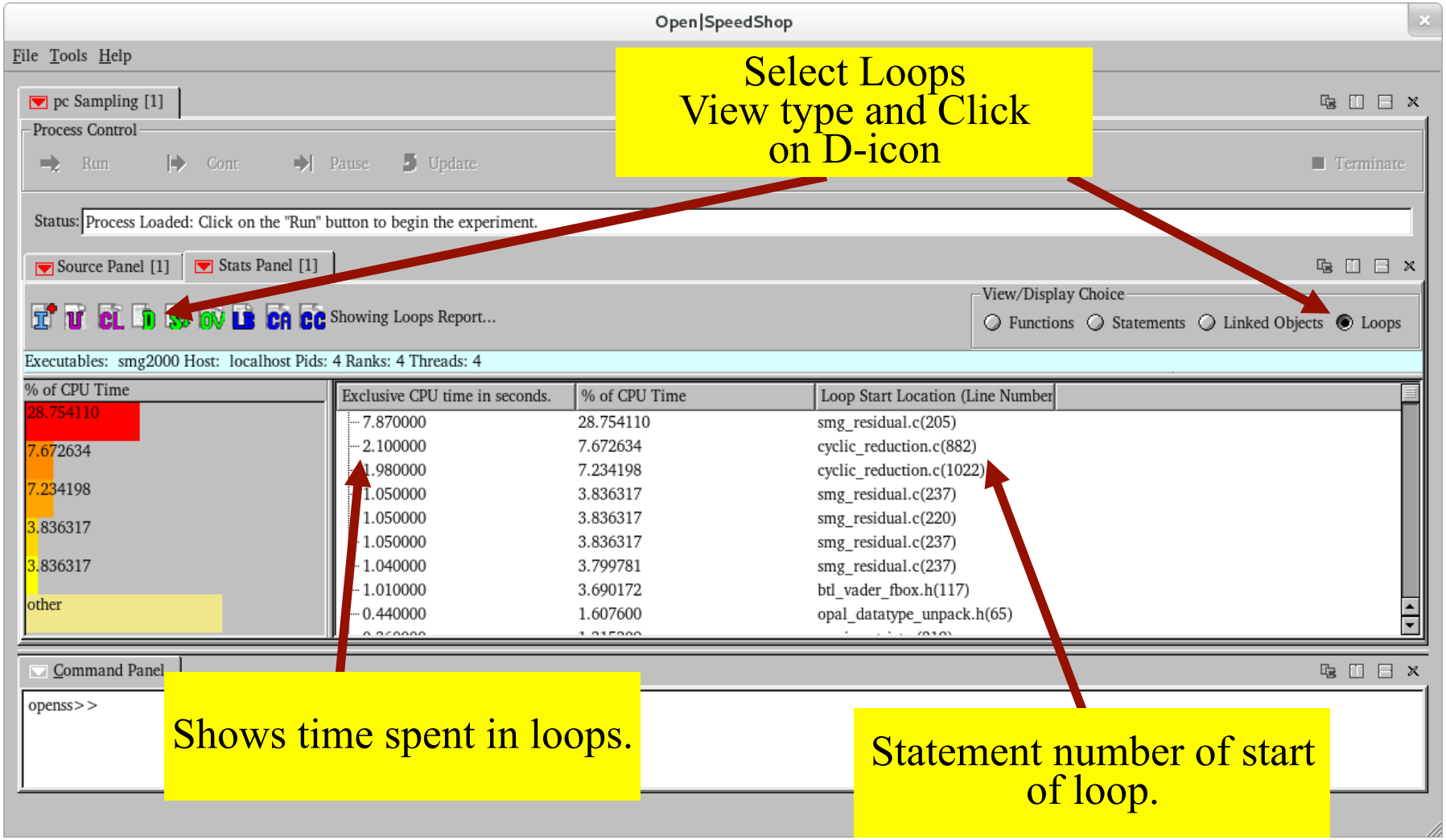
Associate Source & Performance Data



Library (LinkedObject) View



Loop View


Open|SpeedShop
File Tools Help
pc Sampling [1]
Process Control
Run
Cont
Pause
Update
Terminate
Status: Process Loaded: Click on the "Run" button to begin the experiment.
Source Panel [1]
Stats Panel [1]
Showing Loops Report...
View/Display Choice
Functions
Statements
Linked Objects
Loops
Executables: smg2000 Host: localhost Pids: 4 Ranks: 4 Threads: 4
% of CPU Time
28.754110
7.672634
7.234198
3.836317
3.836317
other
Exclusive CPU time in seconds.
% of CPU Time
Loop Start Location (Line Number
7.870000 28.754110 smg_residual.c(205)
2.100000 7.672634 cyclic_reduction.c(882)
1.980000 7.234198 cyclic_reduction.c(1022)
1.050000 3.836317 smg_residual.c(237)
1.050000 3.836317 smg_residual.c(220)
1.050000 3.836317 smg_residual.c(237)
1.040000 3.799781 smg_residual.c(237)
1.010000 3.690172 btl_vader_fbox.h(117)
0.440000 1.607600 opal_datatype_unpack.h(65)
Command Panel
openss>>
Select Loops
View type and Click
on D-icon
Shows time spent in loops.
Statement number of start
of loop.

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 "srun –N 8 –n 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>
- **On clusters, need to set OPENSS_RAWDATA_DIR**
  - Should point to a directory in a shared file system
  - More on this later – usually done in a module or dotkit file.
- **Start with pcsamp for overview of performance**
- **Then, focus on performance issues with other experiments**


SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Section 3

Basic timing experiments and their Pros/Cons





Identifying Critical Regions

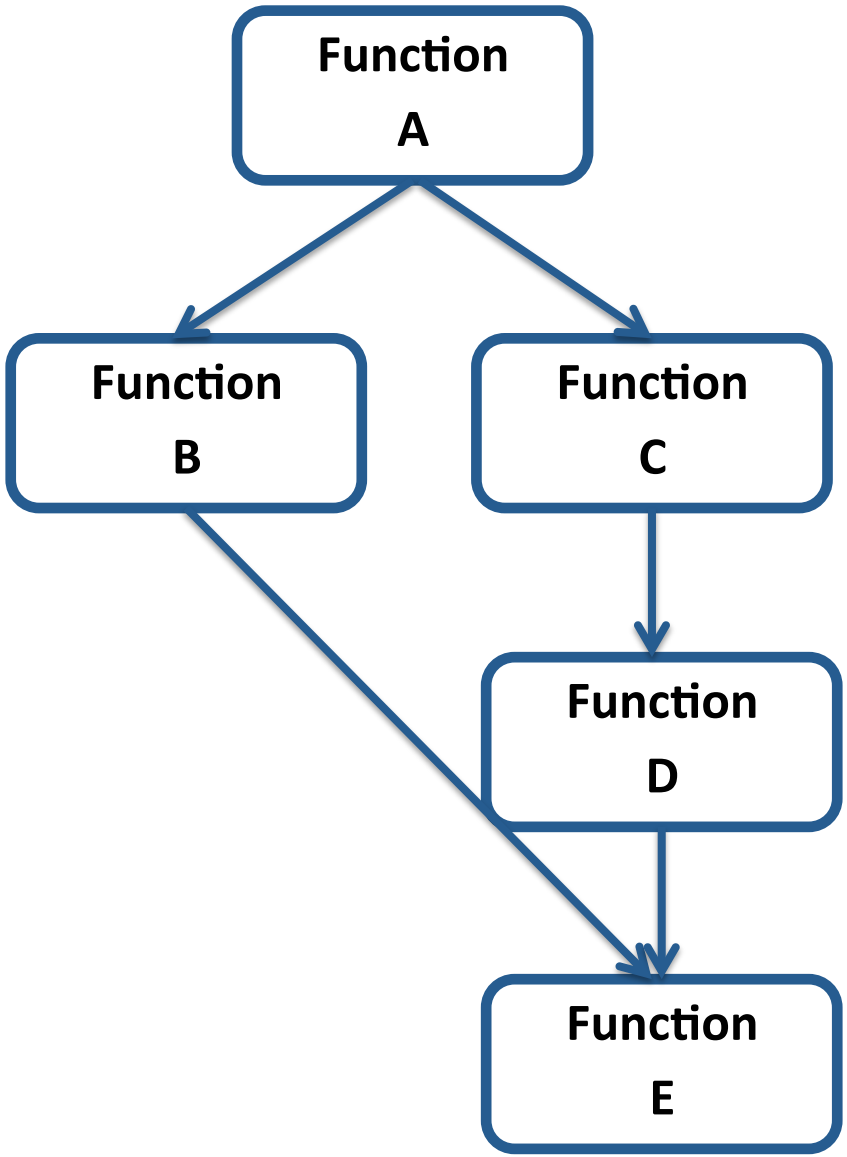
Flat Profile Overview

- **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 stack profiling & Comparisons

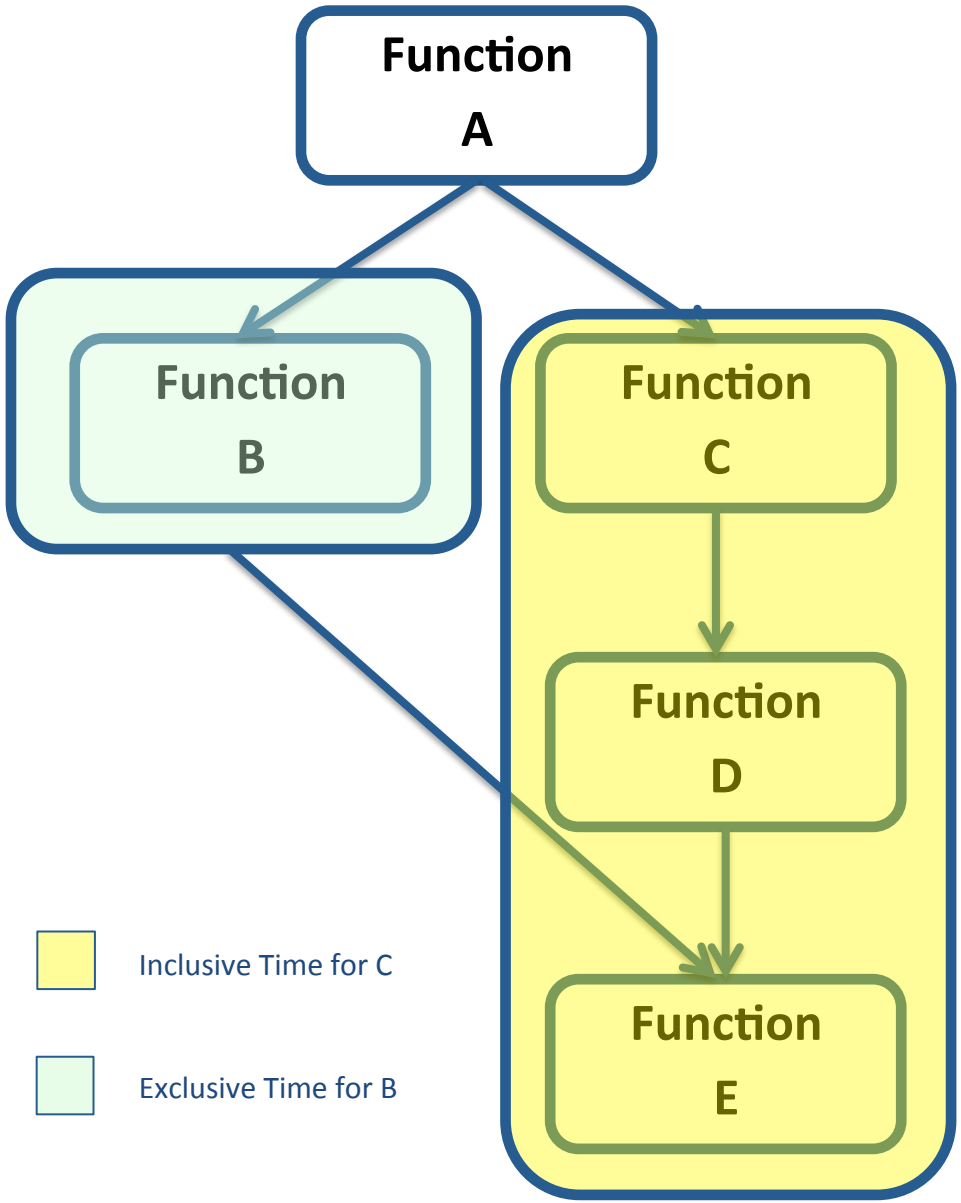
- **Call Stack Profiling**
  - Take a sample: address inside a function
  - Call stack: series of program counter addresses (PCs)
  - Unwinding the stack is walking through those address and recording that information for symbol resolution later.
  - Leaf function is at the end of the call stack list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

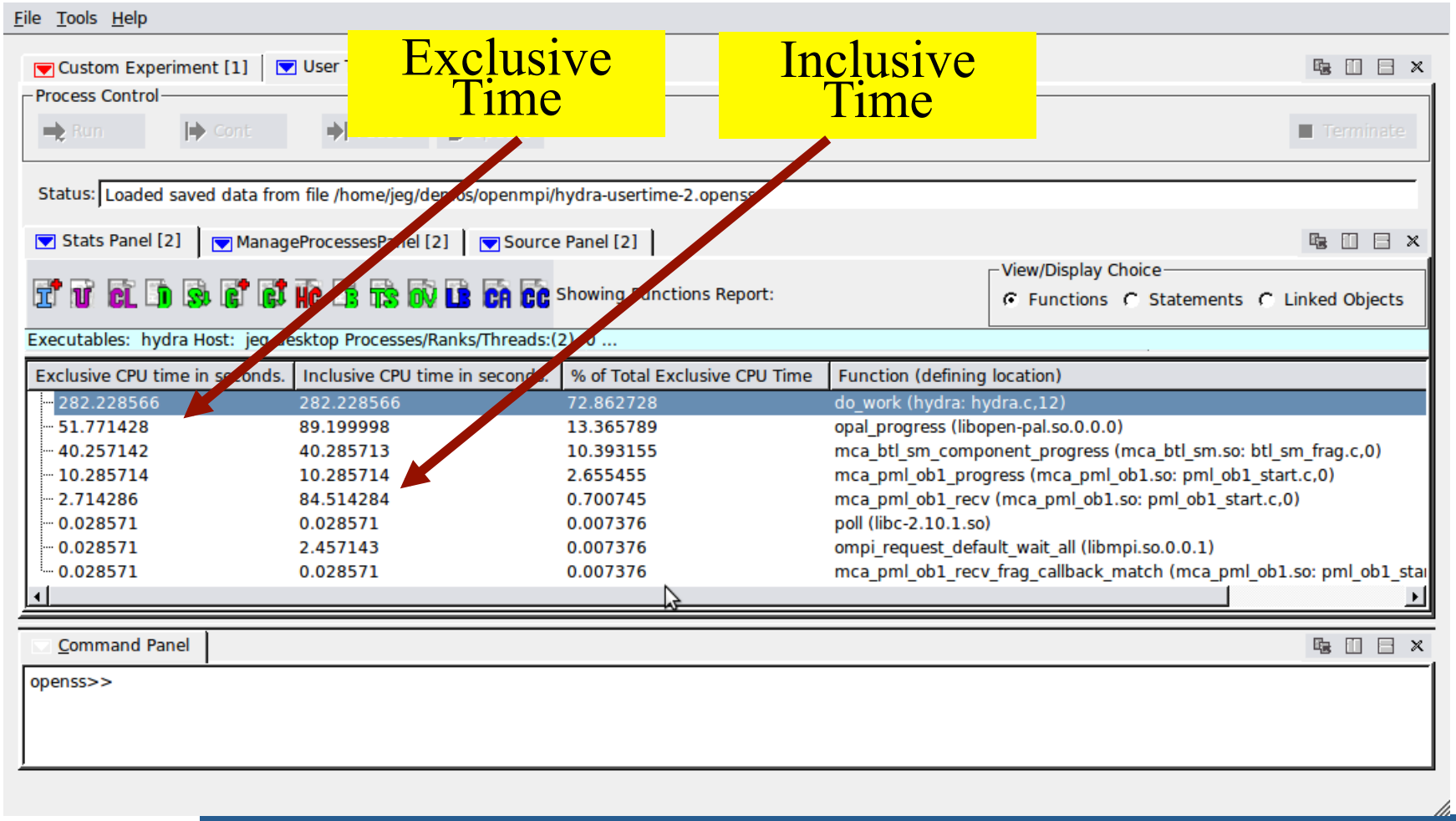
ossusertime “smg2000 –n 50 50 50” low

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

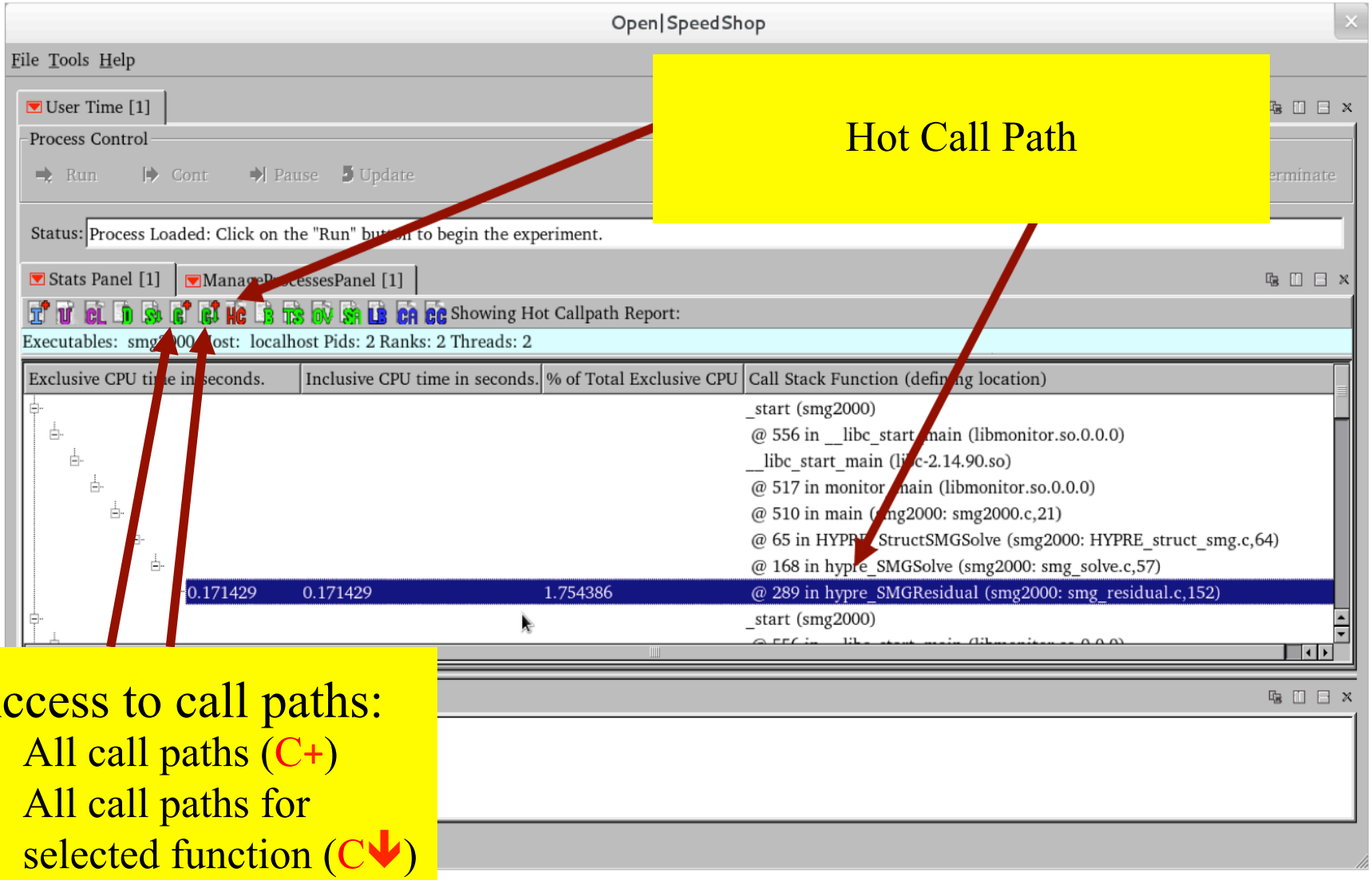
Recommendation: compile code with –g to get statements!

Reading Inclusive/Exclusive Timings

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



Stack Trace Views: Hot Call Path



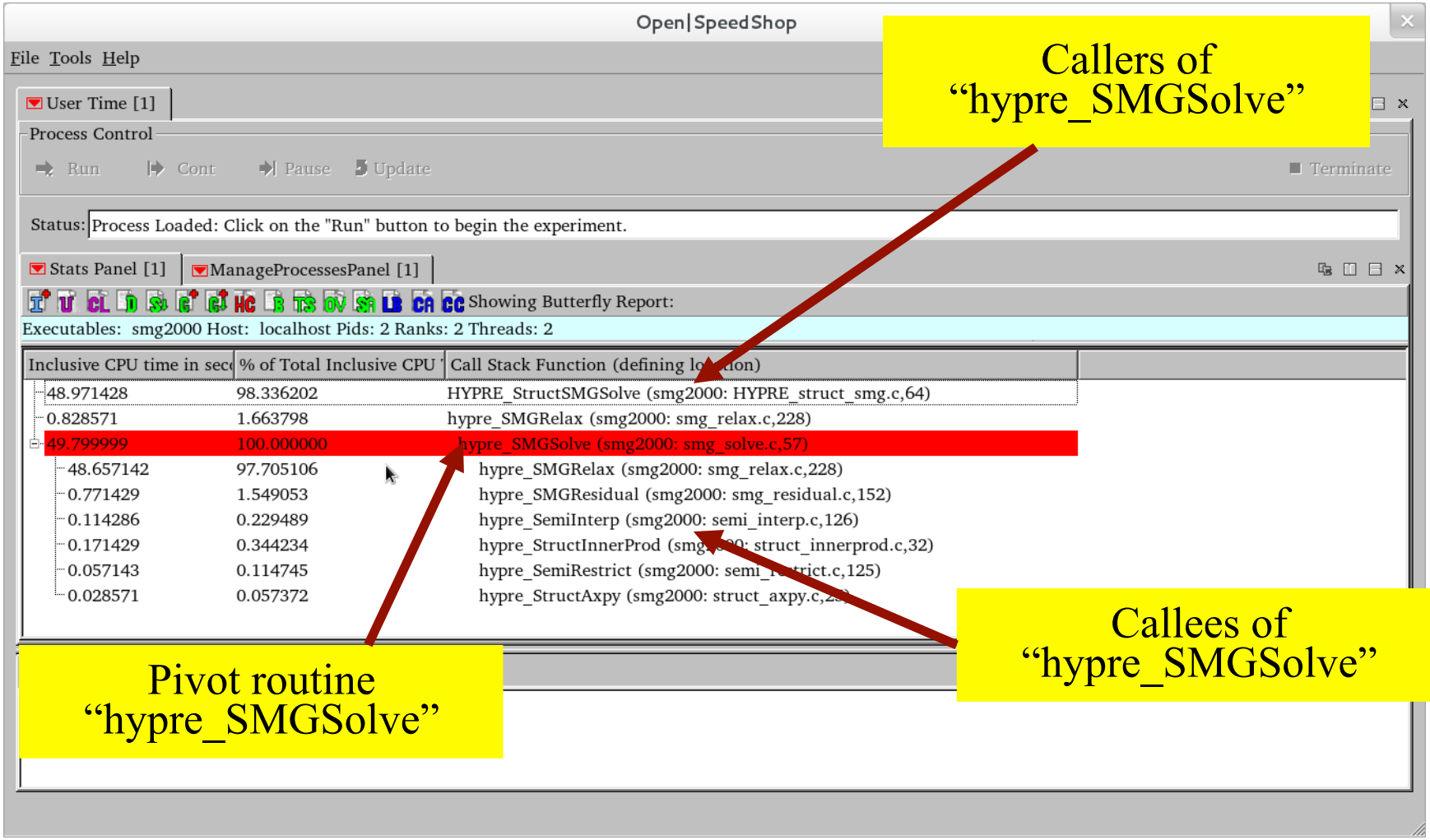
Hot Call Path

Access to call paths:

- All call paths (C+)
- All call paths for selected function (C↓)

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. Function level is the default.
    - 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 time in seconds. | -c 4, Exclusive CPU time in seconds. | Function (defining location)

-c 2, Exclusive CPU time in seconds.	-c 4, Exclusive CPU time in seconds.	Function (defining location)
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	0.040000000	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 base line, then compare against that

SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

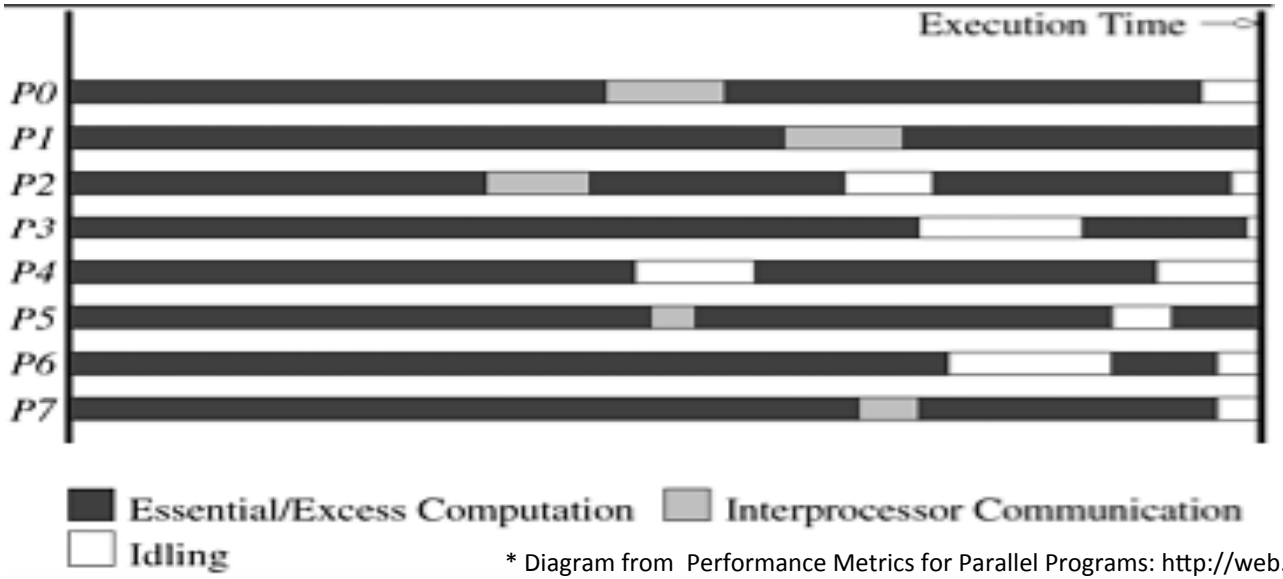
Section 4

Analysis of parallel codes (MPI, threaded)

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



* Diagram from Performance Metrics for Parallel Programs: http://web.info.uvt.ro/~petcu/calcul/

- **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

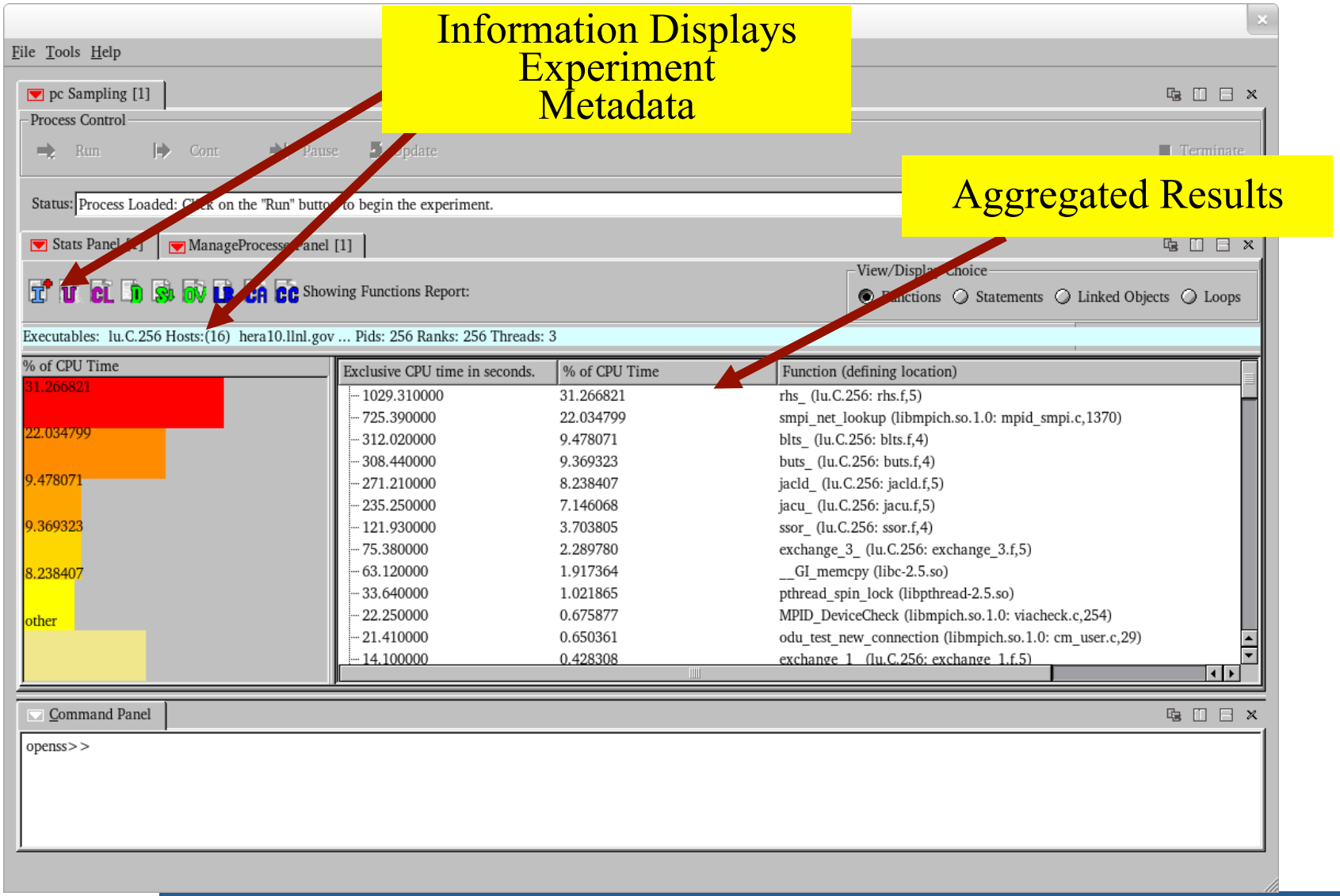
- **What causes the ideal goal to fail?**
  - For MPI:
    - Equal work was not given to each rank
    - There is a out of balance communication pattern occurring
    - The application can't scale with the number of ranks being used
  - For threaded applications:
    - One or more threads doing more 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

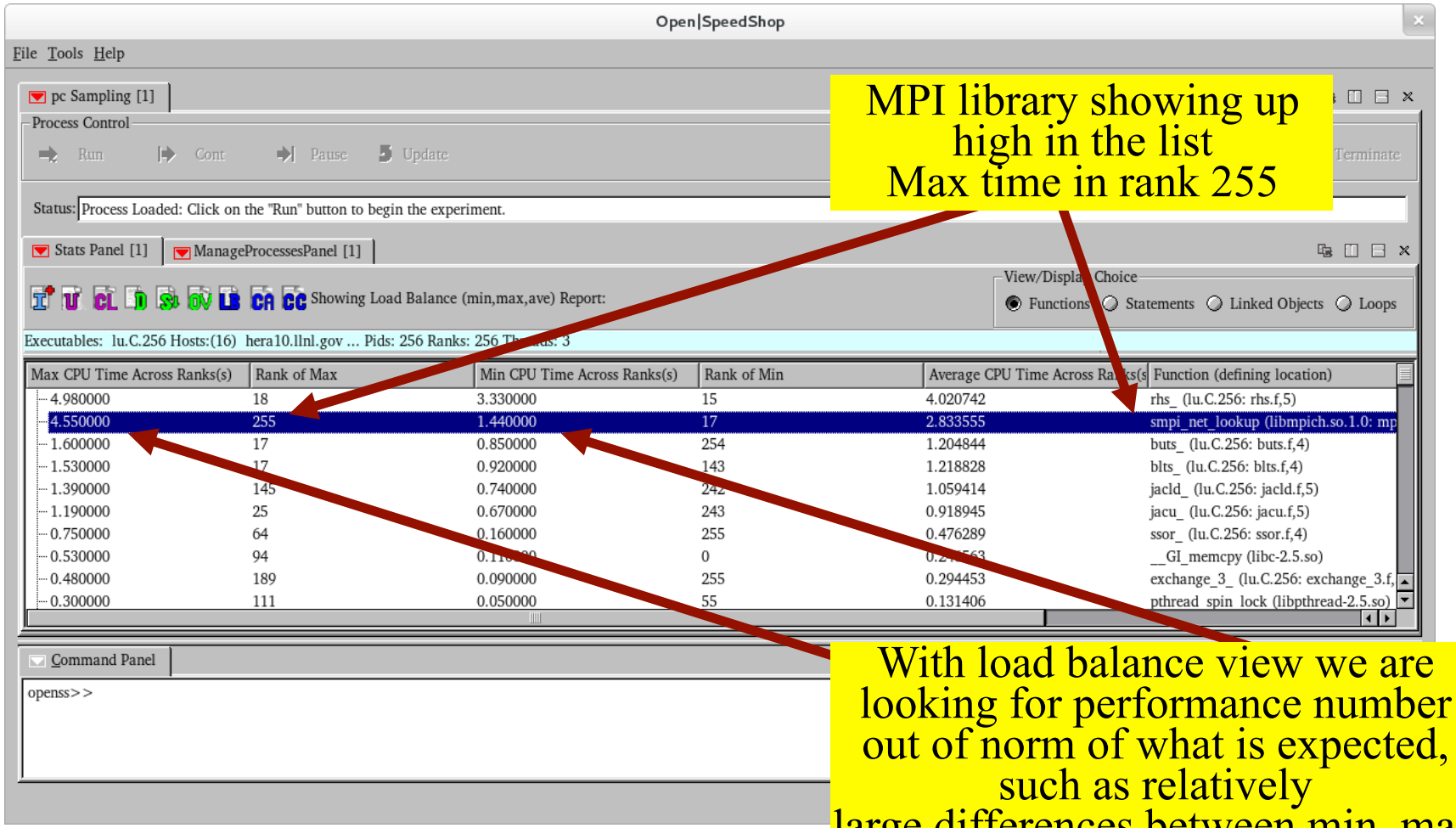
pcsamp Default View: NPB: LU

- **Default Aggregated pcsamp Experiment View**



Load Balance View: NPB: LU

- **Load Balance View based on functions (pcsamp)**

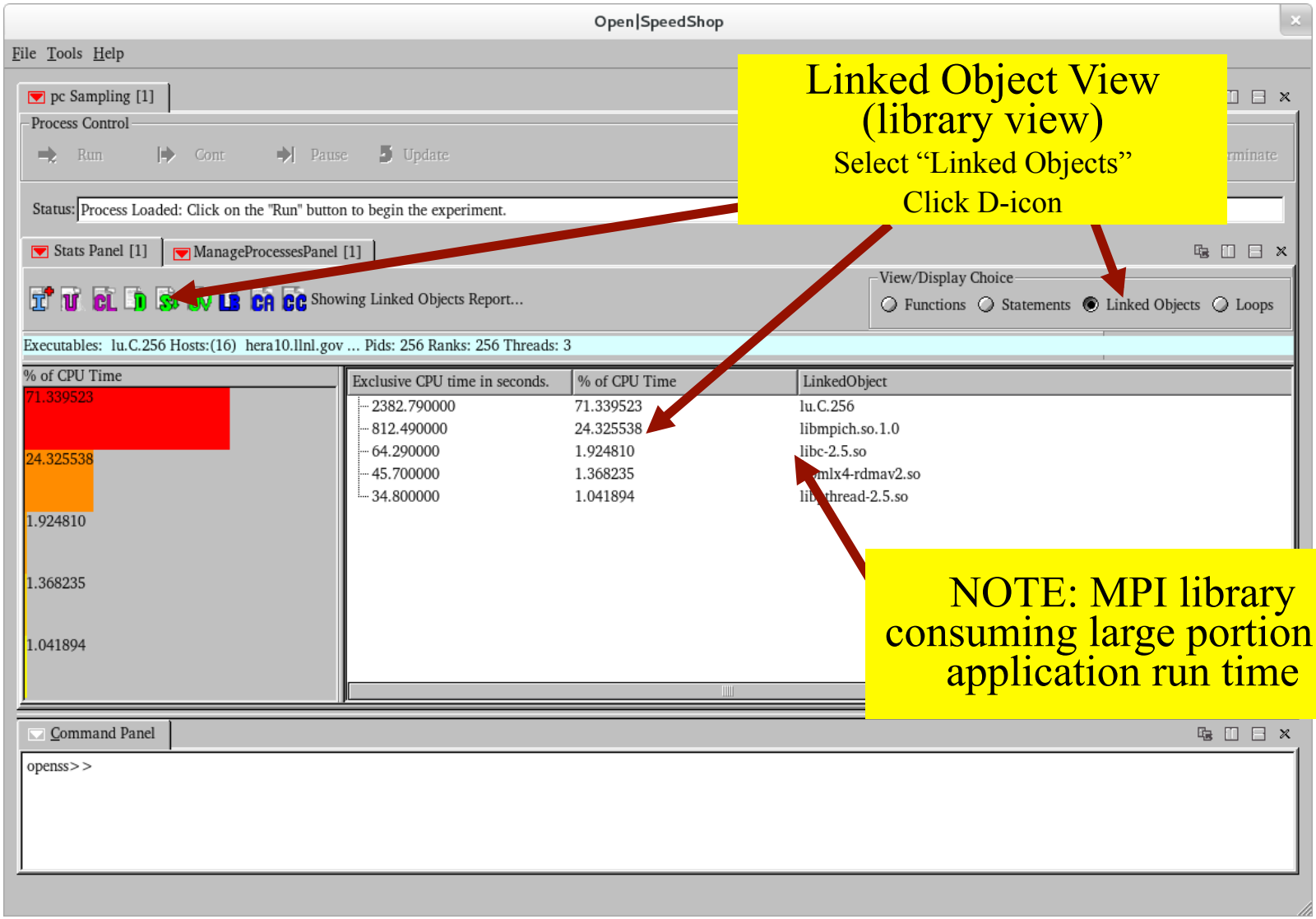


MPI library showing up high in the list
Max time in rank 255

With load balance view we are looking for performance number out of norm of what is expected, such as relatively large differences between min, max and/or average values.

Default Linked Object View: NPB: LU

- **Default Aggregated View based on Linked Objects (libraries)**

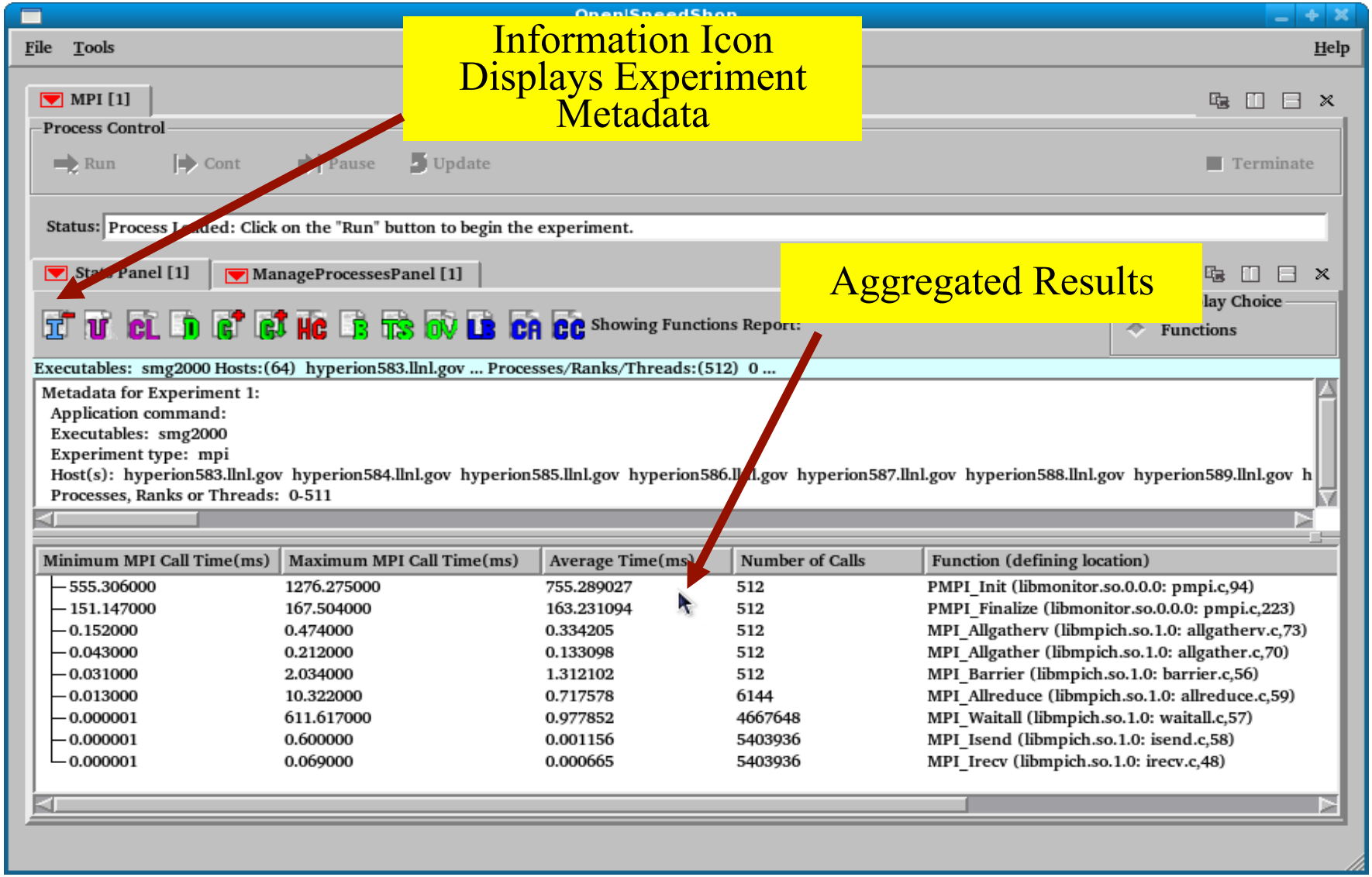


Parallel Execution Analysis Techniques

- **What steps can we take to analyze parallel jobs?**
  - 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 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

MPI Tracing Results: Default View

- **Default Aggregated MPI Experiment View**



Hot Call Paths View (CLI): NPB: LU

❖ Hot Call Paths for MPI_Wait for rank 255 only

```
openss –cli –f lu-mpi-256.openss
openss>>expview -r 255 -vcalltrees,fullstack -f MPI_Wait
```

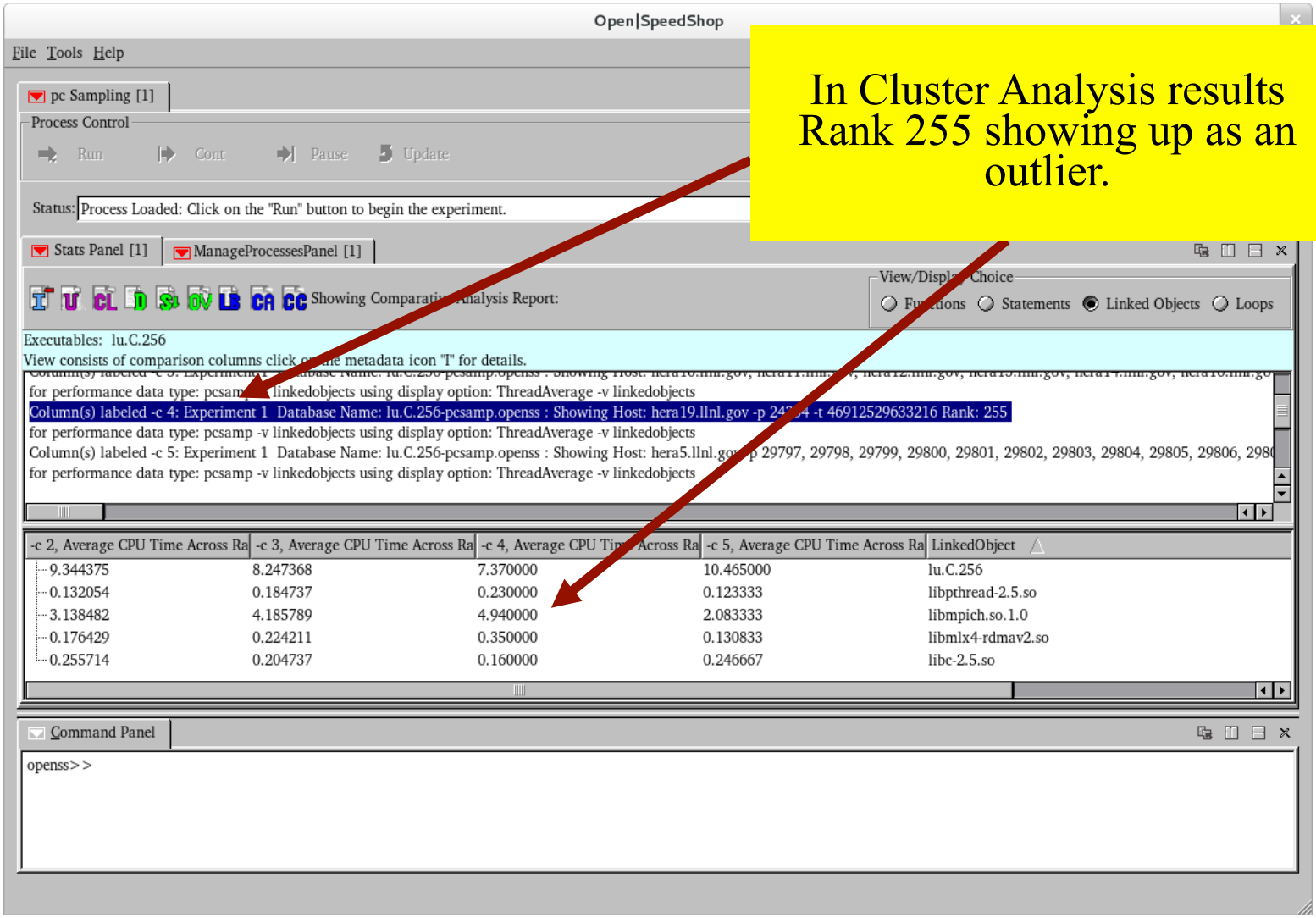
Show all call paths involving MPI_Wait for rank 255 only

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

Most expensive call path to MPI_Wait

Link. Obj. Cluster Analysis: NPB: LU

- **Cluster Analysis View based on Linked Objects (libraries)**



In Cluster Analysis results Rank 255 showing up as an outlier.

MPI Specific Tracing Experiments

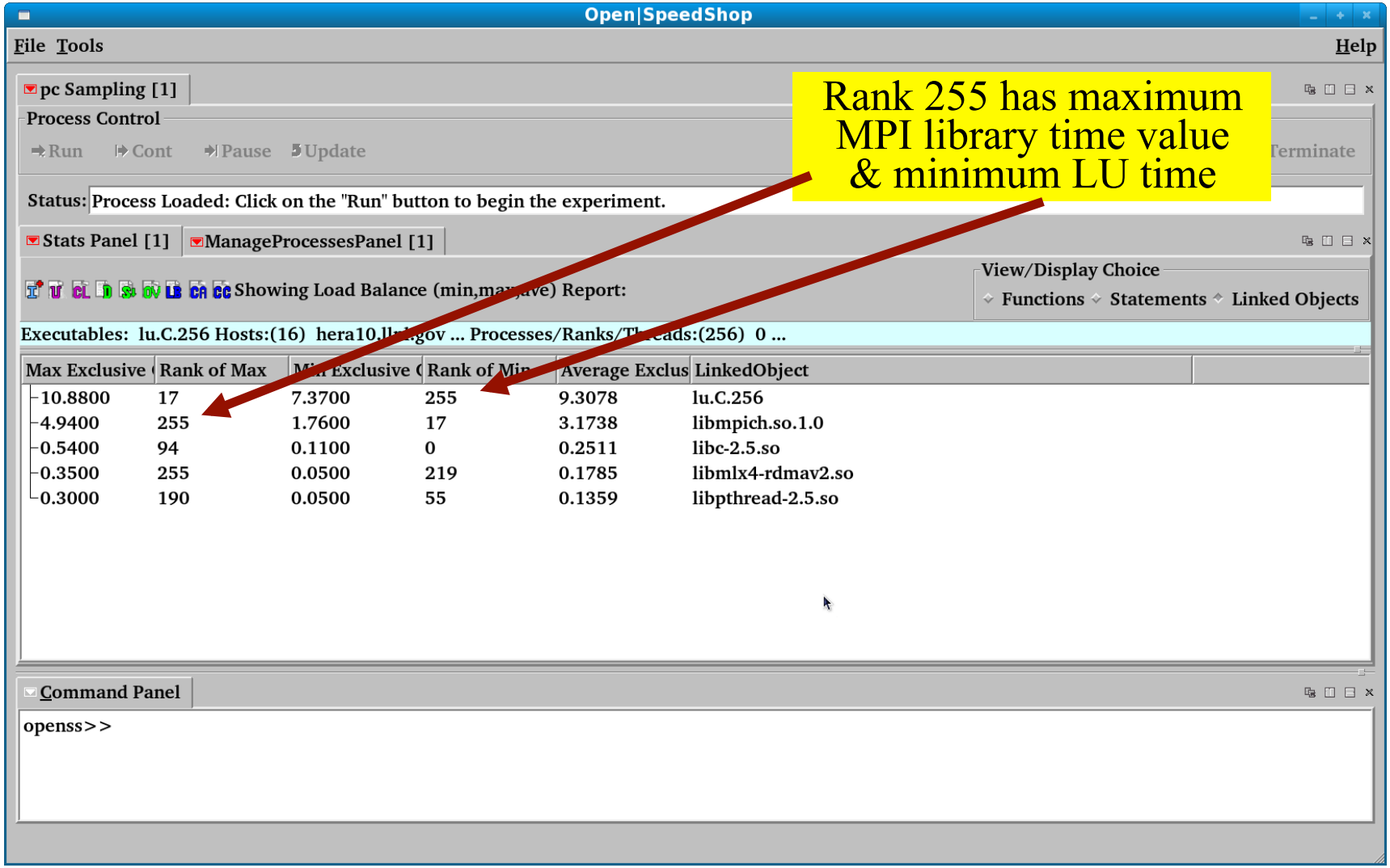
- **MPI function tracing**
  - Record all MPI call invocations
  - Record call times and call paths (**mpi**)
    - Convenience script: ossmpi
  - Record call times, call paths and argument info (**mpit**)
    - Convenience script: ossmpit
- **Equal events will be aggregated**
  - Save space in O|SS database
  - Reduces overhead
- **Public format:**
  - Full MPI traces in Open Trace Format (OTF)
  - Experiment name: (**mpiotf**)
    - Convenience script: ossmpiotf

Identifying Load Imbalance With O|SS

- **Get overview of application**
  - Run one of these lightweight experiments
    - pcsamp, usertime, hwc
  - Use this information to verify performance expectations
- **Use load balance view on pcsamp, usertime, hwc**
  - Look for performance values outside of norm
    - Somewhat large difference for the min, max, average values
- **Get overview of MPI functions used in application**
  - If the MPI libraries are showing up in the load balance for pcsamp, then do a 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

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**
  - 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

Summary / Parallel Bottlenecks

- **Open|SpeedShop supports MPI and threaded application**
  - Works with multiple MPI implementations
- **Parallel experiments**
  - Apply the sequential O|SS collectors to all nodes
  - Specialized MPI tracing experiments
- **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**


SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Section 5

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

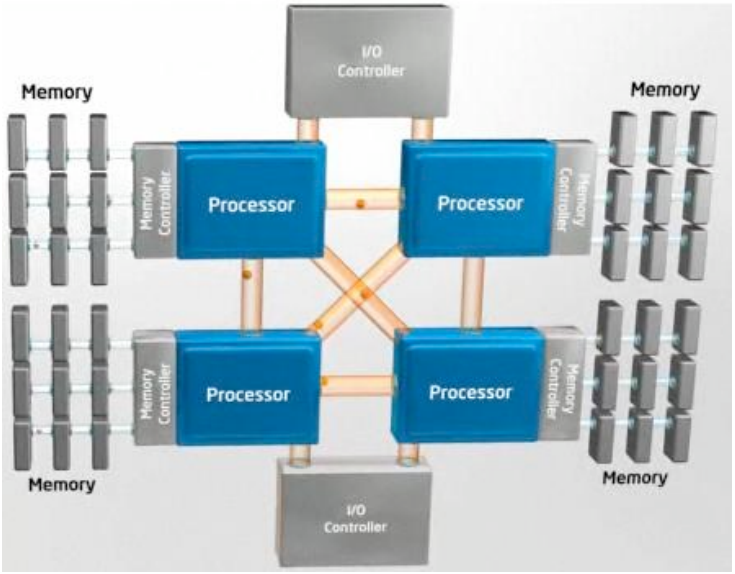
Good Primary Focus: Efficient movement of data

- **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 processor 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

- **osshwcsamp** "<command>< args>" [default | <PAPI_event_list>|<sampling_rate>]
  - 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 cause 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 = 2.28 (kji 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'**

```
do i = 1, n1
  do j = 1, n3
    do k = 1, n2
      a(i,j) = a(i,j) + b(i,k) * c(k,j)
    end do
  end do
end do
```

Metric	IJK	IKJ	JIK	JKI	KIJ	KJI	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)

BLAS Operations Illustrate impact of moving data

A, B, C = nxn Matrices; **x,y** = nx1 Vectors; k = Scaler

Level	Operation	# Memory Refs or Ops	# Flops	Flops/Ops	Comments on Flops/ Ops
1	**y** = k**x** + **y**	3n	2n	2/3	Achieved in Benchmarks
2	**y** = **Ax** + **y**	n^2	$2n^2$	2	Achieved in Benchmarks
3	**C** = **AB** + **C**	$4n^2$	$2n^3$	n/2	Exceeds HW MAX

Use these Flops/Ops to understand how sections of your code relate to simple memory access patterns as typified by these BLAS operations

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, secs	TOT_CYC	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

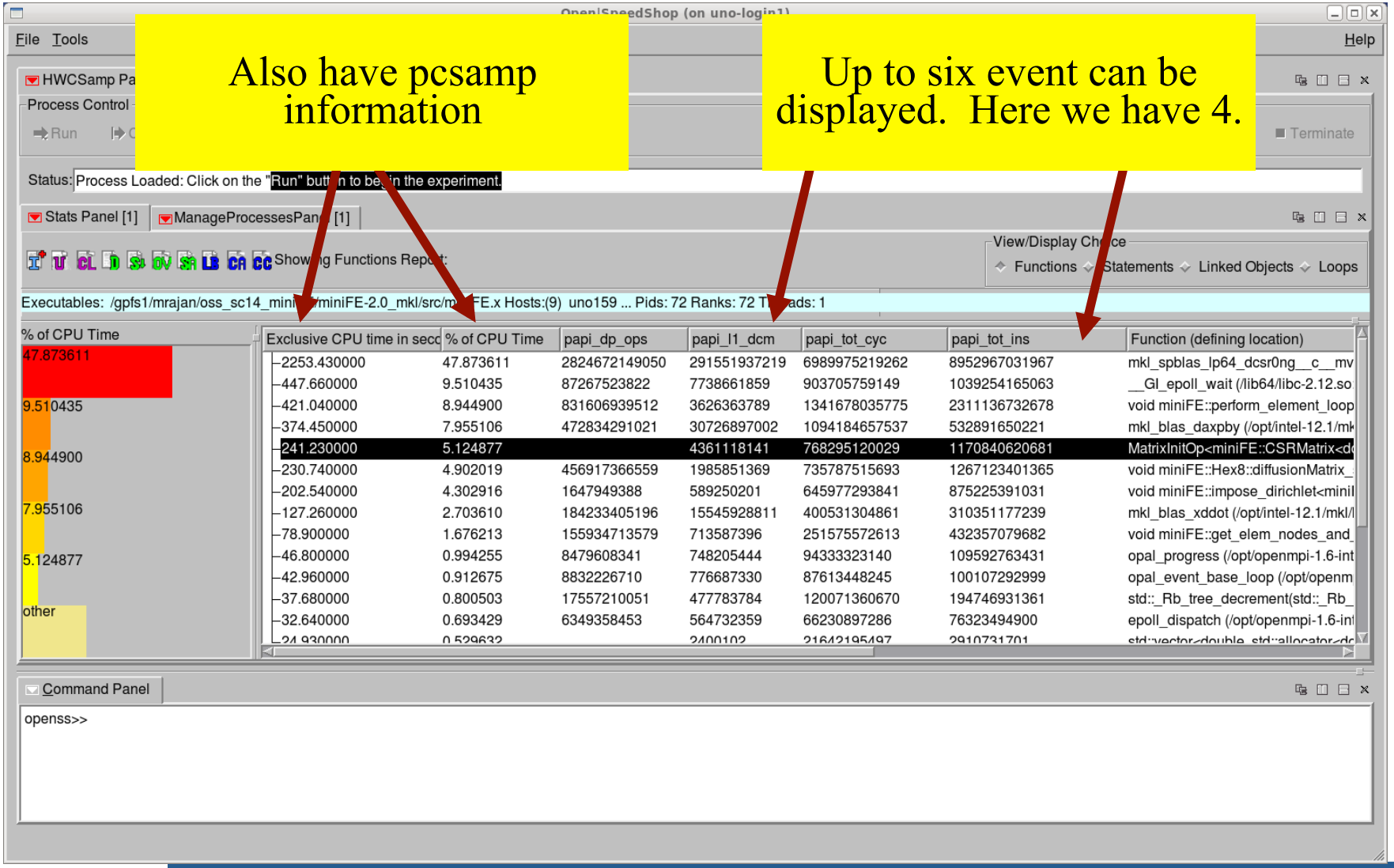
For Intel Haswell FloatOps not available: Use IPC or CPI

- 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

n	Therertical mem refs or Ops	Theoretical FLOP	Theoretical FLOP/Ops	wall time, secs	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	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

mrajan@uno-login1:/gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/run_oss_hwcsamp

File Edit View Search Terminal Help

```
[mrajan@uno-login1 run_oss_hwcsamp]$ openss -cli -f miniFE.x-hwcsamp.openss
openss>>[openss]: The restored experiment identifier is:  -x 1
openss>>expview -v statements hwcsamp30

  Exclusive   % of CPU    papi_dp_ops  papi_l1_dcm   papi_tot_cyc   papi_tot_ins  Statement Location (Line Number)
CPU time in       Time
   seconds.
 447.770000  23.604112    87303886600   7741340402   903892322030  1039416470601  /usr/src/debug/////////glibc-2.12-2-gc4ccff1/misc/../sysdeps/unix/syscall-template.S(82)
 327.880000  17.284133   648522730762   2827873377  1045436496144  1800050381304  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./perform_element_loop.hpp(100)
 199.820000  10.533474                  3806513944   636447396509   965712401233  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./MatrixInitOp.hpp(140)
 180.490000   9.514497      946345017    513583995   575701437696   761402159152  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./SparseMatrix_functions.hpp(475)
  72.920000   3.843964   144552896087    622540931   232540036283   400626778119  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./perform_element_loop.hpp(93)
  67.220000   3.543490   133137191569    576640758   214346202757   369021735348  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../fem/Hex8.hpp(302)
  57.890000   3.051661   114320520328    536923044   184576819581   316915984454  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./Hex8_box_utils.hpp(136)
  35.630000   1.878229                    12642935    27982648181    11492892526  /usr/include/c++/4.4.7/bits/vector.tcc(397)
  25.740000   1.356879    50961487331    222435633    82098246515   141360199733  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../fem/Hex8.hpp(335)
  23.720000   1.250395    46993941906    201176970    75638030490   130325024838  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../fem/Hex8.hpp(330)
  20.090000   1.059041    39815708667    172829164    64060100337   110338757571  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../fem/Hex8.hpp(260)
  19.840000   1.045862     3395917479    296952208    39763200164    46443969250  /builddir/build/BUILD/openmpi-1.6.4/opal/runtime/opal_progress.c(185)
  18.520000   0.976278    36633304484    159683865    59047484493   101653749874  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./perform_element_loop.hpp(82)
  18.150000   0.956774    35947896420    158886460    57884207500    99655374262  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../fem/Hex8.hpp(338)
  16.110000   0.849236    31863068315    140255815    51366606737    88465155762  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../fem/Hex8.hpp(339)
  12.120000   0.638904     2140016350    192110608    24394027282    28522826298  /builddir/build/BUILD/openmpi-1.6.4/opal/event/epoll.c(279)
  11.820000   0.623089                   162934927    37648219524    58301814345  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./MatrixInitOp.hpp(137)
  11.280000   0.594623    22319927660     97114952    35960407509    61927707941  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../fem/Hex8.hpp(336)
  11.170000   0.588824      197905093     34646748    35630595444    54672411406  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./SparseMatrix_functions.hpp(466)
  10.560000   0.556668    14511257762    964669997    32659133238    12804049091  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./cg_solve.hpp(179)
   9.390000   0.494992     1960742963    175257744    19206353960    21857221066  /builddir/build/BUILD/openmpi-1.6.4/opal/event/epoll.c(215)
   9.290000   0.489721                   255848742    27797129336    63963403261  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../utils/box_utils.hpp(297)
   9.270000   0.488666     2017033172    175771745    19021518723    21468715322  /builddir/build/BUILD/openmpi-1.6.4/opal/event/event.c(811)
   8.260000   0.435424                    98650908    26261612341    41367788684  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./MatrixInitOp.hpp(142)
   7.930000   0.418028     1598756218    145174180    16149831865    18544602466  /builddir/build/BUILD/openmpi-1.6.4/opal/runtime/opal_progress.c(206)
   7.870000   0.414866                    61254855    24937401328    59552268291  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./generate_matrix_structure.hpp(105)
   7.780000   0.410121                    36998244    24551069808    59333194907  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./generate_matrix_structure.hpp(115)
   7.210000   0.380074    14227594738     65160484    22988528053    39617523842  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/../fem/Hex8.hpp(300)
   6.430000   0.338956    12745141419     53417855    20499184114    35356808102  /gpfs1/mrajan/oss_sc14_miniFE/miniFE-2.0_mkl/src/./Hex8_box_utils.hpp(147)
   6.210000   0.327359     1319041811    117512799    12639734182    14411185863  /builddir/build/BUILD/openmpi-1.6.4/opal/event/event.c(838)
1668.380000  87.948339  1447432312151  20626804289  4631125982124  6974582554912  Report Summary
openss>>
```

Viewing Data in CLI

Some selections of the powerful CLI commands to view the data

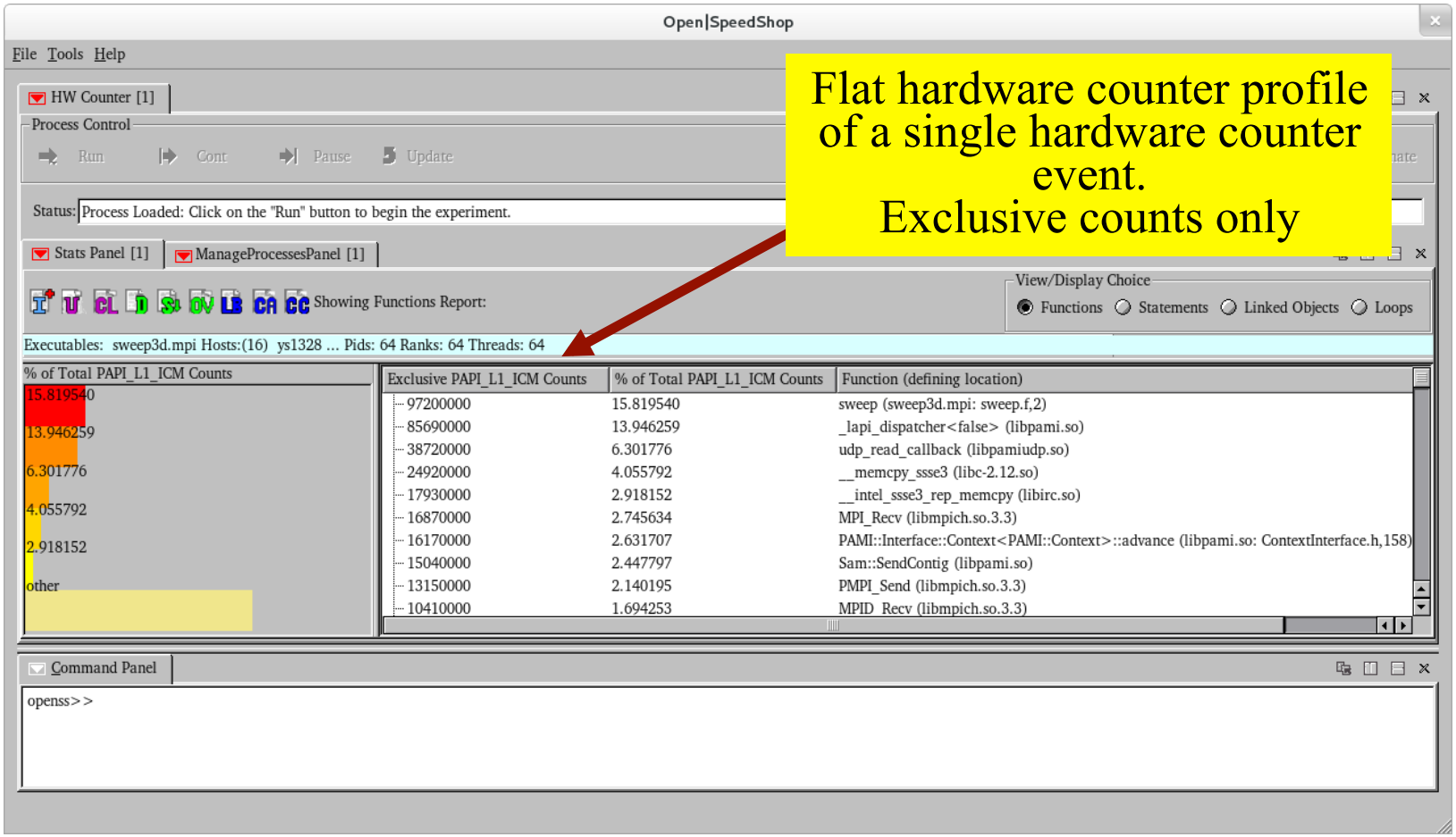
- *expview -v linkedobjects*
- *expview –m loadbalance*
- *expview –v statements hwcsamp*<number>
  - Example to show top 10 statements:
    - *expview –v statements hwcsamp10*
- *expview –v calltrees,fullstack usertime*<number>
- *expcompare – r 1 –r 2 –m time* (compares rank 1 to rank 2 for metric 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 (10000)
- <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

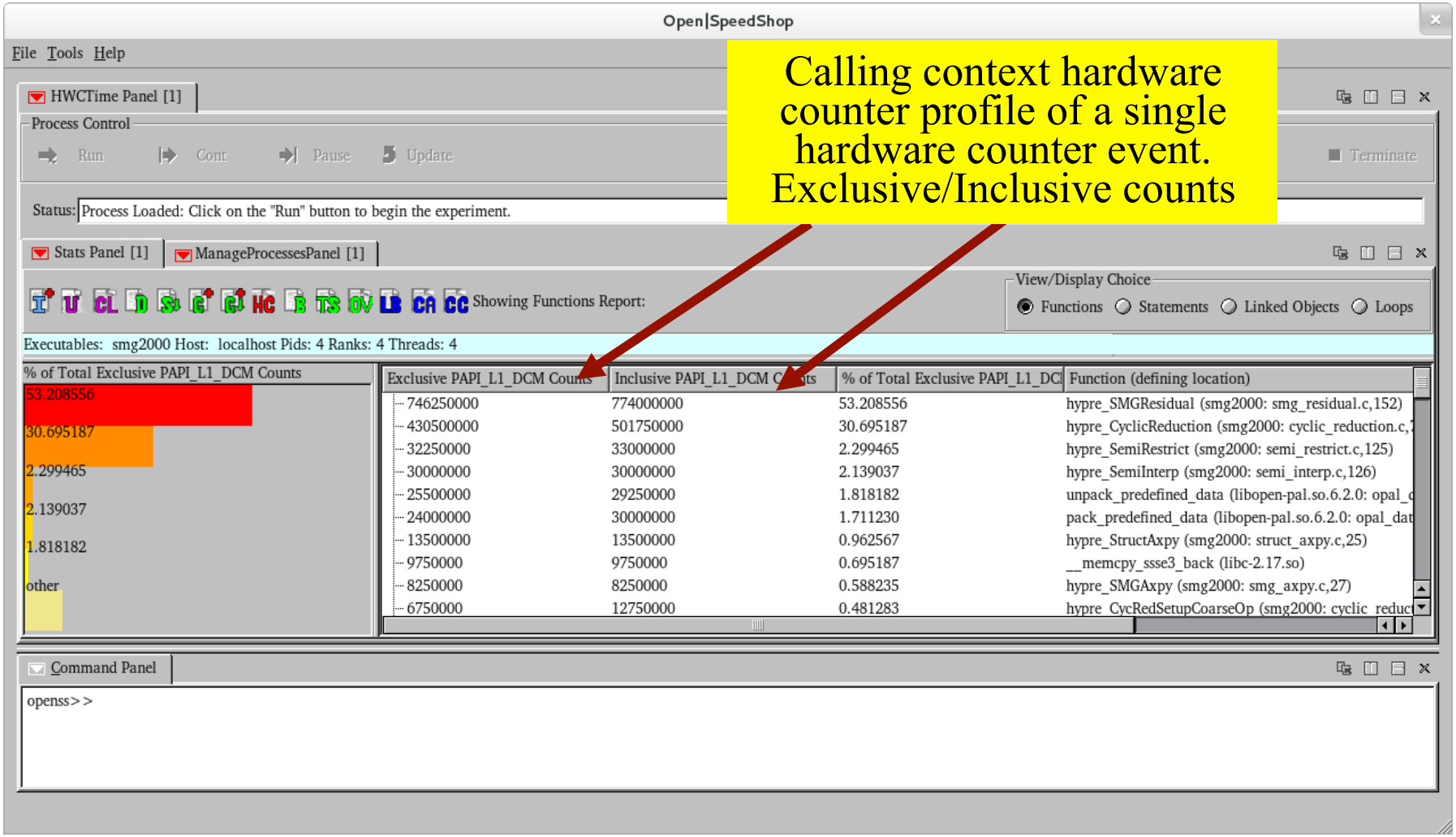
- **hwc default view: Counter = Instruction Cache Misses**



Flat hardware counter profile of a single hardware counter event.
Exclusive counts only

Viewing hwctime Data

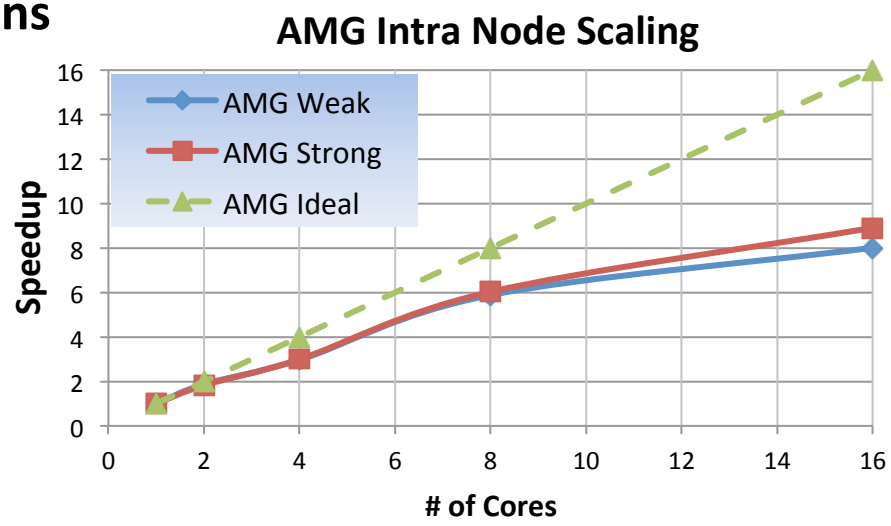
hwctime default view: Counter = L1 Data Cache Misses

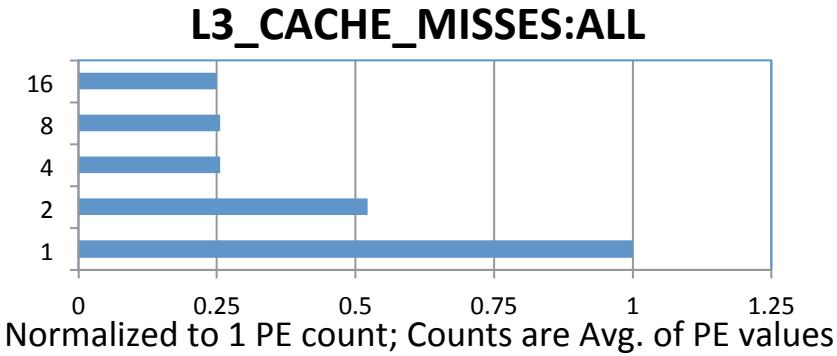


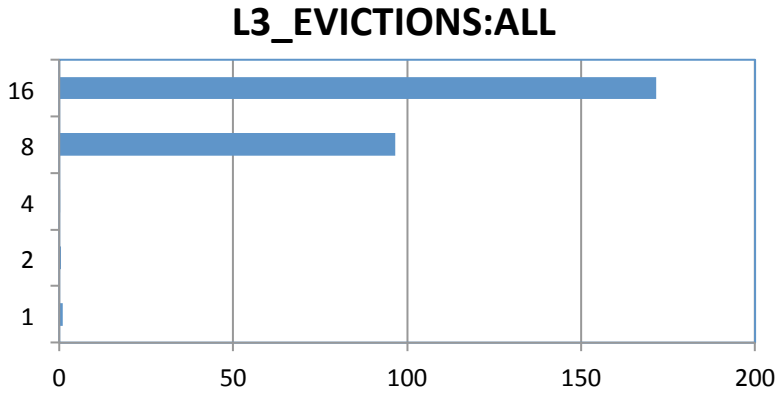
Calling context hardware counter profile of a single hardware counter event. Exclusive/Inclusive counts

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.







- **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

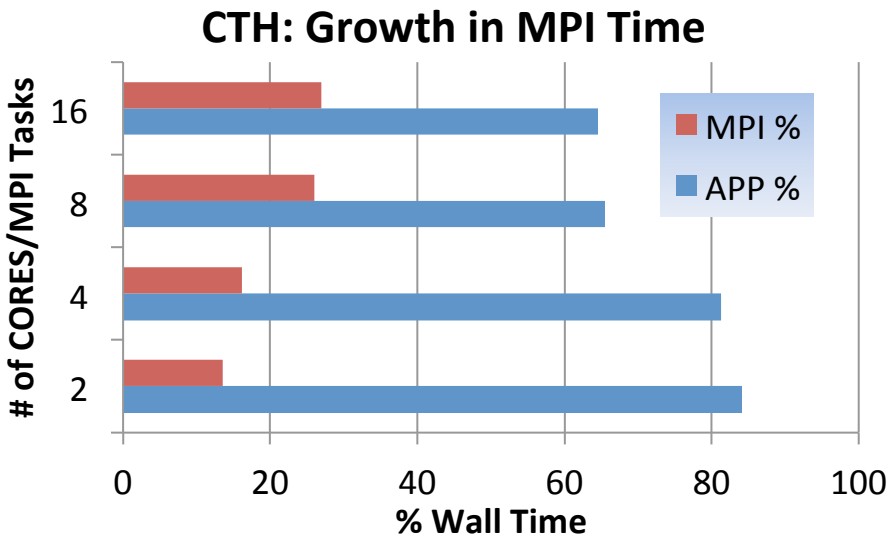
```
! Cache line  UnAligned
real*4, dimension(100,100)::c,d
!$OMP PARALLEL DO
do i=1,100
  do j=2, 100
    c(i,j) = c(i, j-1) + d(i,j)
  enddo
enddo
!$OMP END PARALLEL DO
```

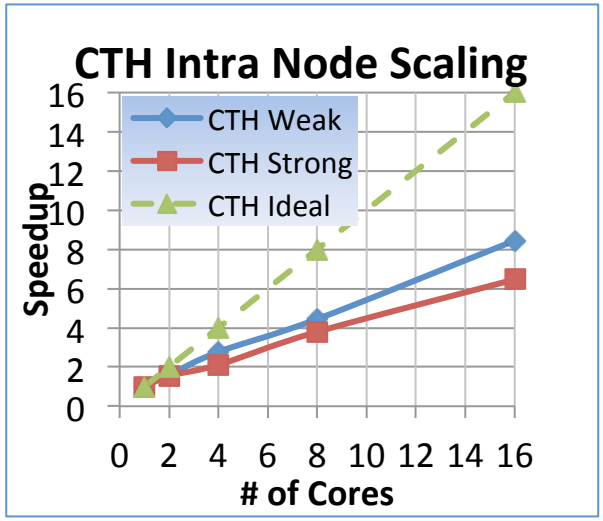
```
! Cache line  Aligned
real*4, dimension(112,100)::c,d
!$OMP DO SCHEDULE(STATIC, 16)
do i=1,100
  do j=2, 100
    c(i,j) = c(i, j-1) + d(i,j)
  enddo
enddo
!$OMP END DO
```

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

Example 3 on use of PAPI: PAPI_TLB_DM Sandia's CTH Performance



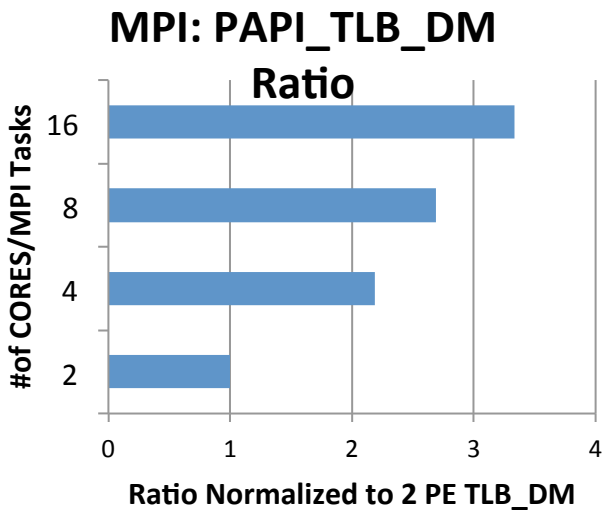


Looking through performance counters the average per PE PAPI counter seeing most increase (among all the profiled functions) with scale is PAPI_TLB_DM registered under MPI. So relinked the executable with –lhugetlbfs, set HUGETLB_MORECORE=yes, and executed with "aprun -m500hs ….."

16 PE performance improvement: 7.35%

128 PE performance improvement: 8.14%

2048 PE performance improvement: 8.23%



SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Section 6
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
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
16	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) = (30999.9999999873,0.000000000000000E+000)

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

sum |xsol_i - x_i|/M = (1.074930753858819E-010,0.000000000000000E+000)

*sum |xsol_i - x_i|/(M*eps) = (968211.548505533,0.000000000000000E+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)

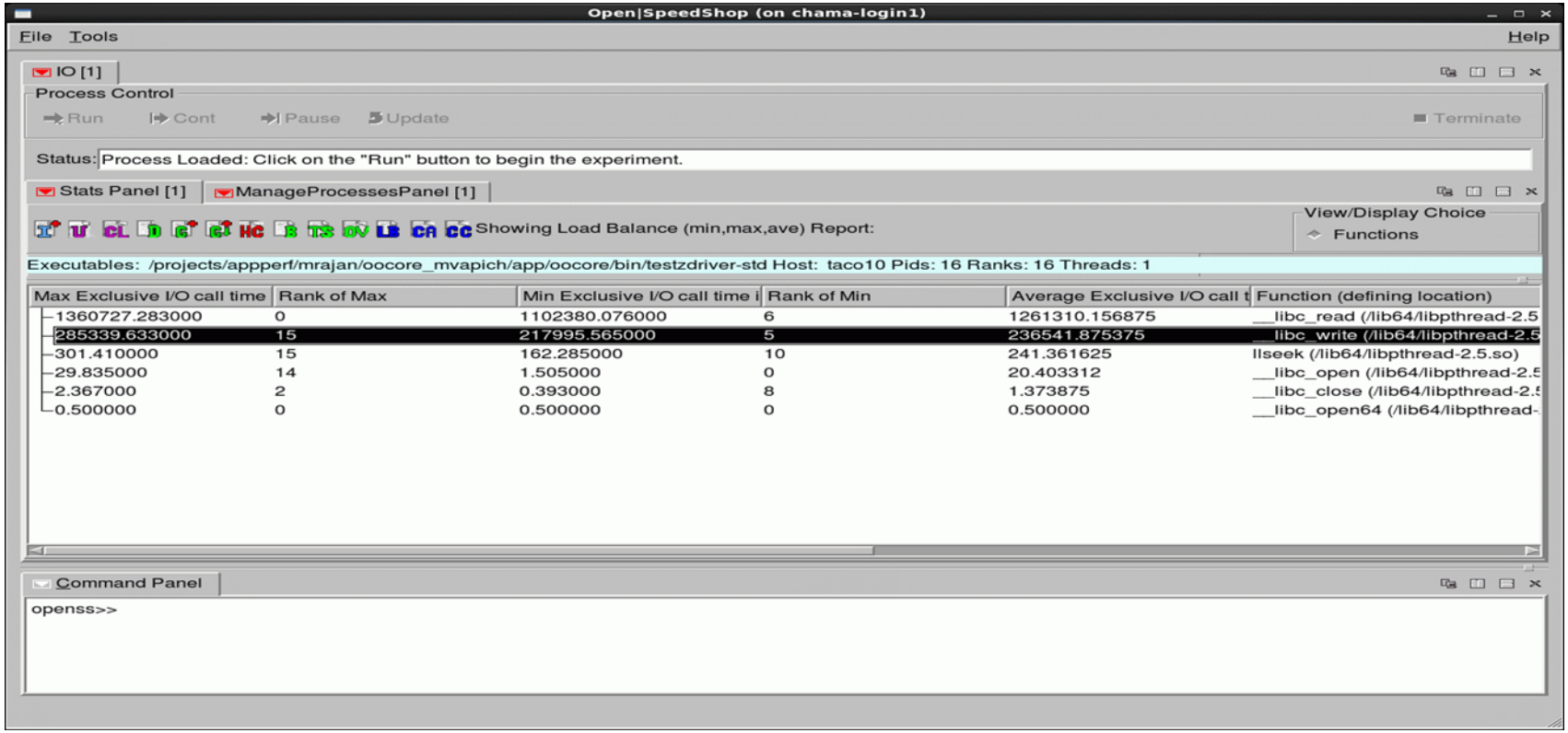
I/O to Lustre instead of NFS reduces runtime 25%: (1360 + 99) – (847 + 7) = 605 secs

NFS RUN

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

LUSTRE RUN

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



Lustre file system striping

Lustre File System (lfs) commands:

lfs 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)

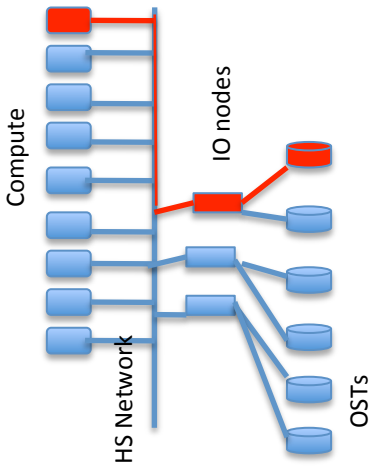
File striping is set upon file creation

lfs getstripe <file | directory>

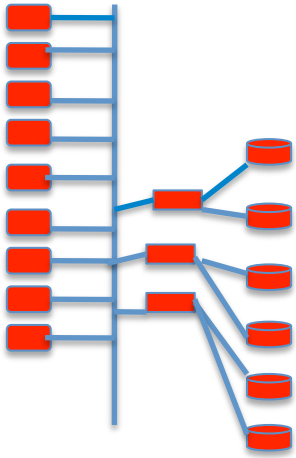
Example: ***lfs getstripe --verbose ./oss_lfs_stripe_16 | grep stripe_count***

stripe_count: 16 stripe_size: 1048576 stripe_offset: -1

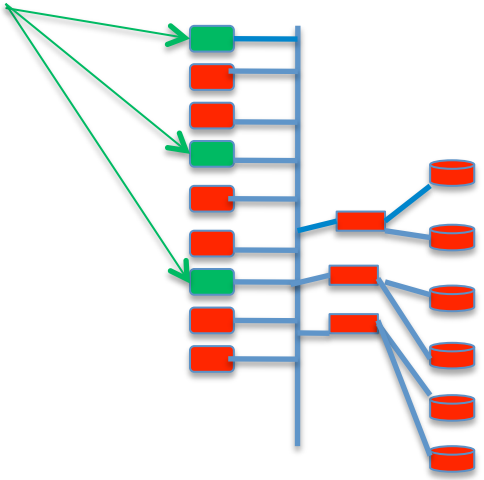
1 PE writes; BW limited



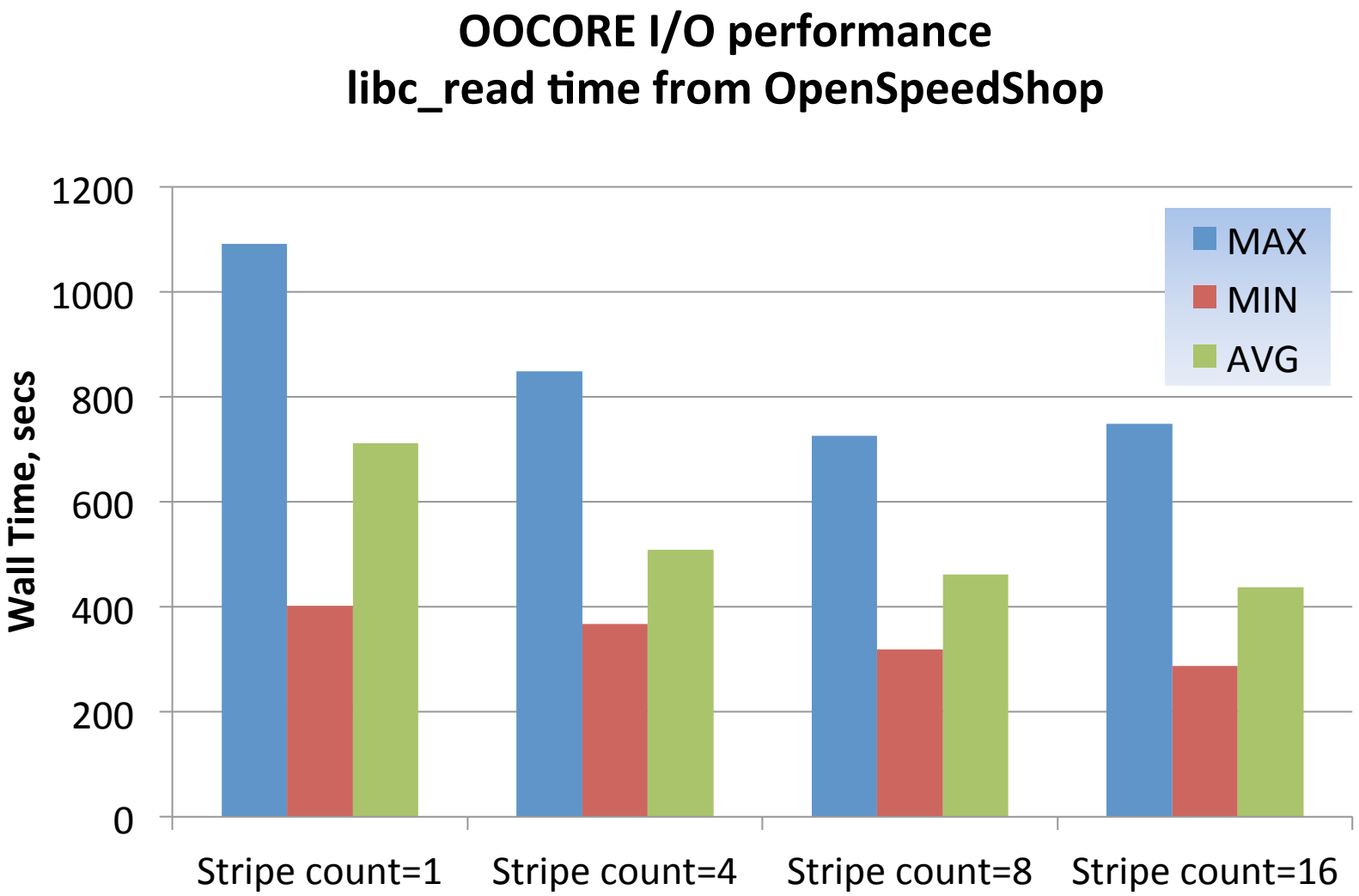
1 file per process; BW enhanced



Subset of PEs do I/O; Could be most optimal



OpenSpeedShop IO-experiment used to identify optimal *lfs striping (from load balance view (max, min & avg) for 16 way parallel run)*



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
{
 int rank;                  // rank of this process within comm
 int size;                  // size of comm
 int tag;                   // for MPI_Send, MPI_Recv
 int baton;                 // for serializing I/O
 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 (cellv, 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);
    cellv = (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

Open | SpeedShop

File Tools Help

Custom Experiment [1]

Process Control

Run | Cont | Pause | Update | Terminate

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] | ManageProcessesPanel [1]

I U CL D G GI HC B TS OV EL LB CA CC Showing Per Event Report:

View/Display Choice

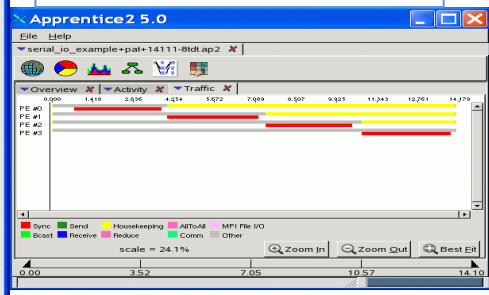
Functions

Executables: (none) Host: glory238 Processes/Ranks/Threads:(4) 0 ...

Start Time	I/O Call Time(ms)	% of Total Time	Call Stack Function (defining location)
2010/09/08 13:22:54	0.029000	6.682028	>__libc_read (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.026000	5.990783	>__libc_write (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.008000	1.843318	>__libc_read (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.058000	13.364055	>__libc_write (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.061000	14.055300	>__libc_write (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.010000	2.304147	>__libc_read (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.016000	3.686636	>__libc_read (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.015000	3.456221	>__libc_read (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.025000	5.760369	>__libc_read (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.021000	4.838710	>__libc_read (/lib64/libpthread-2.5.so)
2010/09/08 13:22:54	0.015000	3.456221	>__libc_write (/lib64/libpthread-2.5.so)

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:

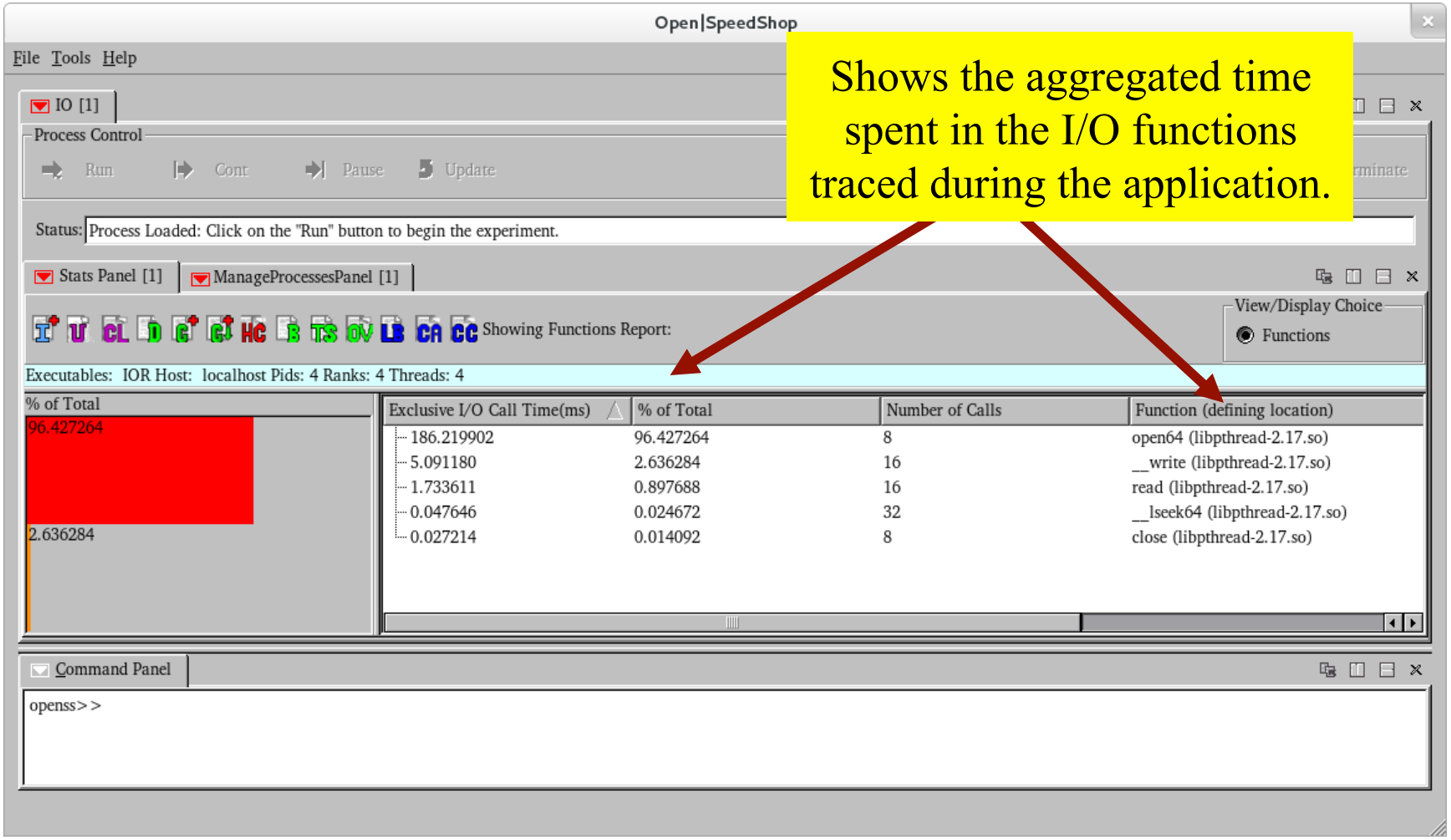
ossio “mpirun –np 256 sweep3d.mpi”

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

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

I/O output via GUI

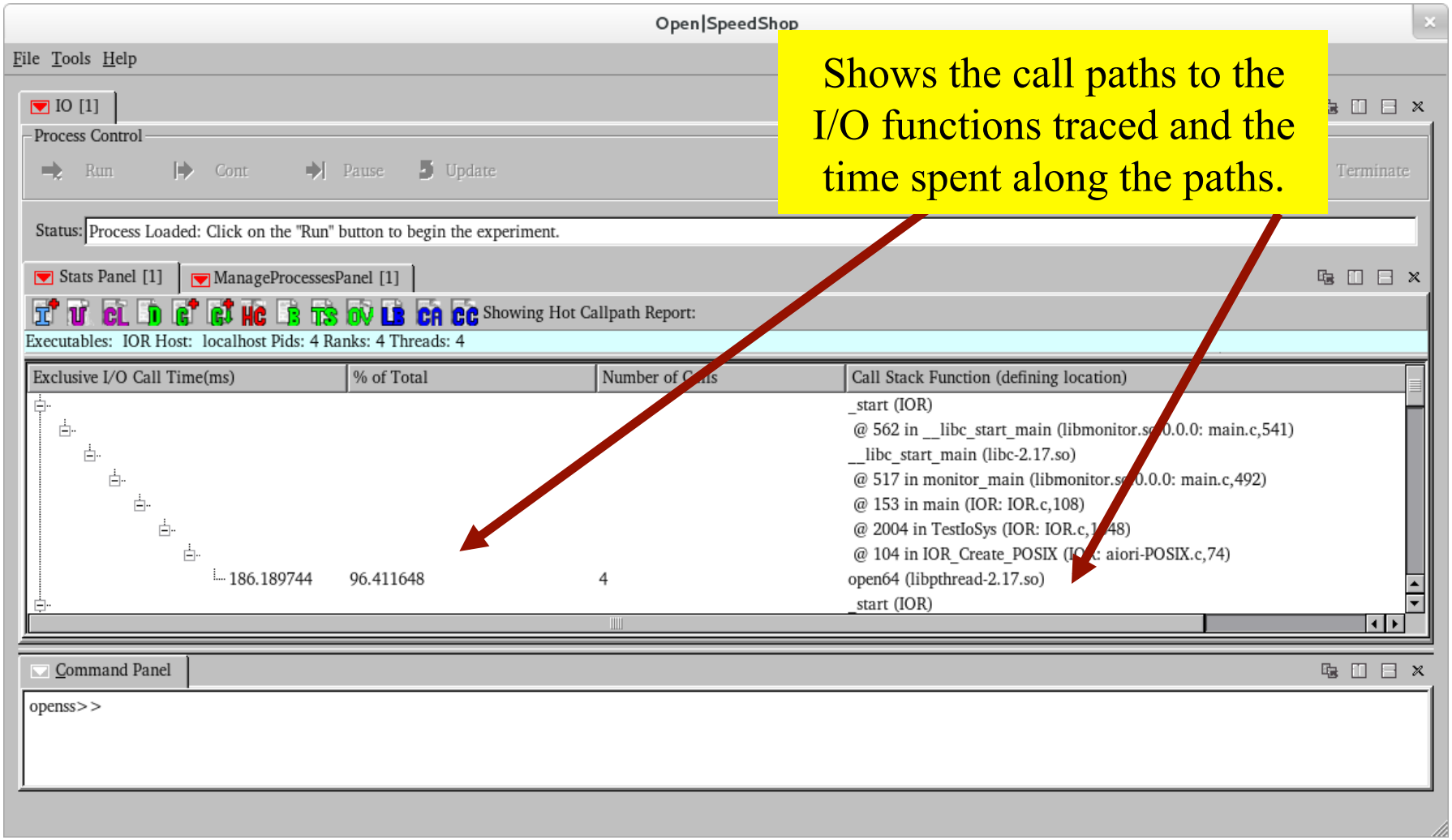
❖ I/O Default View for IOR application “io” experiment



Shows the aggregated time spent in the I/O functions traced during the application.

I/O output via GUI

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



Shows the call paths to the I/O functions traced and the time spent along the paths.

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

SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Section 7
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 highwater mark (HWM).
  - List of event allocations (with calling context) to leaks.
  - Overview of all unique calltrees to traced memory calls that provides max and min allocation and count of calls on this path.
- **Example Usage**
  - ossmem "./lulesh2.0”
  - ossmem "srun -N4 -n 64 ./sweep3d.mpi"

O|SS Memory Experiment

- **Show calltree for the allocation event that set the high water mark**

```
openss>> expview -v trace,calltrees -m start_time -m hwm -m allocation
Start Time of Event      HWM      Call Stack Function (defining location)
                                      _start (lulesh2.0)
                                      >__libc_start_main (libmonitor.so.0.0.0: main.c,541)
                                      >>__libc_start_main (libc-2.18.so)
                                      >>>monitor_main (libmonitor.so.0.0.0: main.c,492)
                                      >>>>main (lulesh2.0: lulesh.cc,2672)
                                      >>>>>GOMP_parallel_start (libgomp.so.1.0.0: parallel.c,123)
                                      >>>>>>gomp_new_team (libgomp.so.1.0.0: team.c,141)
                                      >>>>>>>gomp_malloc (libgomp.so.1.0.0: alloc.c,35)
2015/09/09 10:11:04.023  27016684 >>>>>>>>__libc_malloc (libc-2.18.so)
```

O|SS Memory Experiment

❖ Show calltree to one memory leak

expview -v trace,calltrees -m leak -m allocation mem1

Total Call Stack Function (defining location)

Allocation

_start (lulesh2.0)

>__libc_start_main (libmonitor.so.0.0.0: main.c,541)

>>__libc_start_main (libc-2.18.so)

>>>monitor_main (libmonitor.so.0.0.0: main.c,492)

>>>>main (lulesh2.0: lulesh.cc,2672)

>>>>>GOMP_parallel_start (libgomp.so.1.0.0: parallel.c,123)

>>>>>>gomp_new_team (libgomp.so.1.0.0: team.c,141)

>>>>>>>gomp_malloc (libgomp.so.1.0.0: alloc.c,35)

10600684 >>>>>>>>__libc_malloc (libc-2.18.so)

O|SS Memory Experiment

- **The next slide shows an overview of unique call paths to memory calls.**
- **In this example:**
  - The top free and malloc calls are shown with number of times the path was called.
- **For the malloc call:**
  - The max and min allocation seen on this path are displayed.

O|SS Memory Experiment

```
expview -v trace,calltrees,fullstack -mpath_counts -m max_allocation,min_allocation -m overview
Counts       Max       Min       Call Stack Function (defining location)
 This  Allocation  Allocation
 Path
                                 _start (lulesh2.0)
                                 > @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
                                 >>__libc_start_main (libc-2.18.so)
                                 >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
                                 >>>> @ 2392 in main (lulesh2.0: lulesh.cc,2672)
                                 >>>>> @ 2249 in EvalEOSForElems(Domain&, double*, int, int*, int)
(lulesh2.0: lulesh.cc,2218)
 32619                           >>>>>>__cfree (libc-2.18.so)
                                 _start (lulesh2.0)
                                  > @ 562 in __libc_start_main (libmonitor.so.0.0.0: main.c,541)
                                  >>__libc_start_main (libc-2.18.so)
                                  >>> @ 517 in monitor_main (libmonitor.so.0.0.0: main.c,492)
                                  >>>> @ 2392 in main (lulesh2.0: lulesh.cc,2672)
                                  >>>>> @ 2164 in EvalEOSForElems(Domain&, double*, int, int*, int)
(lulesh2.0: lulesh.cc,2218)
                                 >>>>>> @ 125 in GOMP_parallel_start (libgomp.so.1.0.0: parallel.c,123)
                                 >>>>>>> @ 156 in gomp_new_team (libgomp.so.1.0.0: team.c,141)
                                 >>>>>>>> @ 37 in gomp_malloc (libgomp.so.1.0.0: alloc.c,35)
 32619      2080      2080      >>>>>>>>>__libc_malloc (libc-2.18.so)
```

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 generate large amounts of data
- ➢ Can slow down and impact application while running

SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Section 8

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
    - Vector units, SIMD execution
    - Single instruction operates on multiple data items
  - Thread level parallelism
    - 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 language or use directive based OpenCL or 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

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

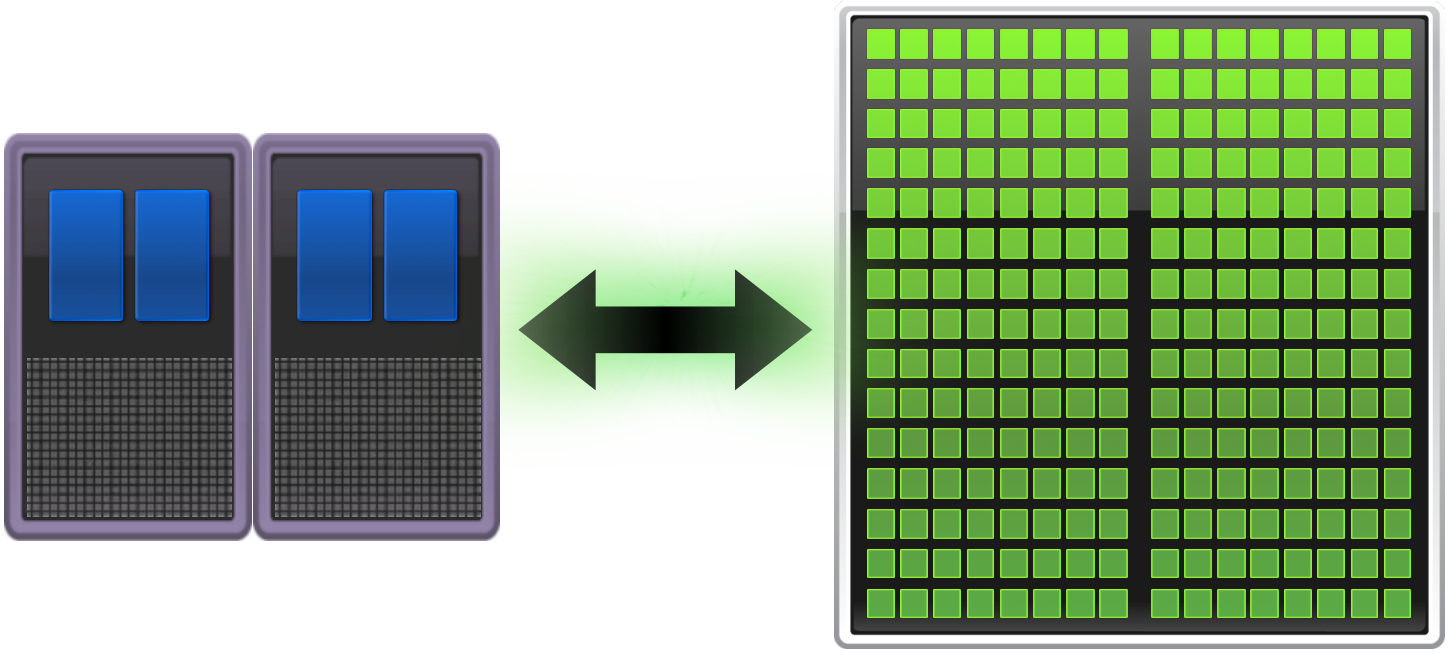
- **GPU: maximize throughput of all threads**
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - Multi-threading can hide latency => skip the big caches
  - Shared control logic across many threads

***based on NVIDIA presentation**

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

Prominent models for programming the GPGPU

Augment current languages to access GPU strengths

- NVIDIA CUDA
  - Scalable parallel programming model
  - Minimal 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 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 MIC 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 will be 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 scheduled**
- **OpenCL support is also scheduled (after OpenACC)**

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 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 experiment is coming soon
    - Will help to better support analysis of MIC applications
    - OpenMP performance analysis key to understanding performance

Open|SpeedShop CUDA CLI Views

```
>openss –cli -f BlackScholes-cuda-0.openss
Welcome to OpenSpeedShop 2.1
[openss]: The restored experiment identifier is:  -x 1
openss>>expview
 Exclusive        % of Total        Exclusive  Function (defining location)
 Time (ms)   Exclusive Time        Count
170.478194  100.000000       131  BlackScholesGPU(float*, float*, float*, float*, float*, float, float, int) (BlackScholes)
openss>>expview -v Exec,Trace
Start Time (d:h:m:s)  Exclusive   % of Total
Grid Dims  Block Dims  Call Stack Function (defining location)
                              Time (ms)  Excl Time
2014/12/04 00:47:18.139   1.305890   0.766016  31250,1,1  128,1,1  >BlackScholesGPU(float*, float*, float*, float*, float*, float, float, int) (BlackScholes)
2014/12/04 00:47:18.141   1.297314   0.760985  31250,1,1  128,1,1  >BlackScholesGPU(float*, float*, float*, float*, float*, float, float, int) (BlackScholes)
…
2014/12/04 00:47:18.805   1.300547   0.762882  31250,1,1  128,1,1  >BlackScholesGPU(float*, float*, float*, float*, float*, float, float, int) (BlackScholes)
```

CUDA CLI Views

[openss>>expview -v Xfer,Trace

Start Time (d:h:m:s)	Exclusive Time (ms)	% of Total Exclusive Time	Size	Kind	Call Stack Function (defining location)
2014/12/04 00:47:18.127	4.060334	14.766928	16000000	HostToDevice	>>cudart::cudaApiMemcpy(void*, void const*, unsigned long, cudaMemcpyKind) (BlackScholes)
2014/12/04 00:47:18.131	3.882354	14.119637	16000000	HostToDevice	>>cudart::cudaApiMemcpy(void*, void const*, unsigned long, cudaMemcpyKind) (BlackScholes)
2014/12/04 00:47:18.135	3.893426	14.159904	16000000	HostToDevice	>>cudart::cudaApiMemcpy(void*, void const*, unsigned long, cudaMemcpyKind) (BlackScholes)
2014/12/04 00:47:18.817	8.012517	29.140524	16000000	DeviceToHost	>>cudart::cudaApiMemcpy(void*, void const*, unsigned long, cudaMemcpyKind) (BlackScholes)
2014/12/04 00:47:18.825	7.647501	27.813007	16000000	DeviceToHost	>>cudart::cudaApiMemcpy(void*, void const*, unsigned long, cudaMemcpyKind) (BlackScholes)

openss>>expview -v Exec,Trace -m time,grid,block,rpt

Exclusive Time (ms)	Grid Dims	Block Dims	Registers Per Thread	Call Stack Function (defining location)
1.305890	31250,1,1	128,1,1	15	>BlackScholesGPU(float*, float*, float*, float*, float*, float, float, int) (BlackScholes)
1.304930	31250,1,1	128,1,1	15	>BlackScholesGPU(float*, float*, float*, float*, float*, float, float, int) (BlackScholes)
…				
1.295267	31250,1,1	128,1,1	15	>BlackScholesGPU(float*, float*, float*, float*, float*, float, float, int) (BlackScholes)

CUDA Trace View (Chronological order)

openss>>expview -v trace

Start Time(d:h:m:s)	Exclusive Call Time(ms)	% of Total	Call Stack Function (defining location)
2013/08/21 18:31:21.611	11.172864	1.061071	copy 64 MB from host to device (CUDA)
2013/08/21 18:31:21.622	0.371616	0.035292	copy 2.1 MB from host to device (CUDA)
2013/08/21 18:31:21.623	0.004608	0.000438	copy 16 KB from host to device (CUDA)
2013/08/21 18:31:21.623	0.003424	0.000325	set 4 KB on device (CUDA)
2013/08/21 18:31:21.623	0.003392	0.000322	set 137 KB on device (CUDA)
2013/08/21 18:31:21.623	0.120896	0.011481	compute_degrees(int*, int*, int, int)<<<[256,1,1], [64,1,1]>>> (CUDA)
2013/08/21 18:31:21.623	13.018784	1.236375	QTC_device(float*, char*, char*…,int, bool)<<<[256,1,1], [64,1,1]>>> (CUDA)
2013/08/21 18:31:21.636	0.035232	0.003346	reduce_card_device(int*, int)<<<[1,1,1], [1,1,1]>>> (CUDA)
2013/08/21 18:31:21.636	0.002112	0.000201	copy 8 bytes from device to host (CUDA)
2013/08/21 18:31:21.636	1.375616	0.130640	trim_ungrouped_pnts_indr_array(int, int*, float*…,float, int, bool)<<<[1,1,1], [64,1,1]>>> (CUDA)
2013/08/21 18:31:21.638	0.001344	0.000128	copy 260 bytes from device to host (CUDA)
2013/08/21 18:31:21.638	0.025600	0.002431	update_clustered_pnts_mask(char*, char*, int)<<<[1,1,1], [64,1,1]>>> (CUDA)

Open|SpeedShop Accelerator support

What is the status of this Open|SpeedShop work?

- **Completed, but not hardened:**
  - NVIDIA GPU Performance Data Collection
  - Packaging the data into the Open|SpeedShop database
  - Retrieving the performance data and forming text based views
  - Basic initial support for Intel Phi (MIC) based applications.
    - Have gathered and displayed data on MIC test bed at NERSC and at NASA on the maia Intel MIC platform.
- **On going work (NASA FY15-FY16 SBIR)**
  - Pursue obtaining more information about the code being executing inside the GPU device
  - More research and hardening of support for Intel MIC
  - Analysis of performance data and more meaningful views for CUDA/GPU
  - Research creating GUI based views based on initial command line interface (CLI) data views.


SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study 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 and PowerPC64
  - 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 MIC support available
  - Available on many DOE/DOD systems in shared locations
  - Ask you 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**
  - Two parts of the installation
    - Krell Root – base packages
    - O|SS 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: 2.2 has been released**
- **Open|SpeedShop Website**
  - http://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
- **Sourceforge Project Home**
  - http://sourceforge.net/projects/openss
- **CVS Access**
  - http://sourceforge.net/scm/?type=cvs&group_id=176777
- **Packages**
  - Accessible from Project Home Download Tab
  - Also accessible from www.openspeedshop.org

Open|SpeedShop Documentation

- **Build and Installation Instructions**
  - http://www.openspeedshop.org/documentation
    - Look for: Open|SpeedShop Version 2.2 Build/Install Guide
- **Open|SpeedShop User Guide Documentation**
  - http://www.openspeedshop.org/documentation
    - Look for Open|SpeedShop Version 2.2 Users Guide
- **Man pages: OpenSpeedShop, osspcsamp, ossmpi, …**
- **Quick start guide downloadable from web site**
  - http://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
  - Lot's 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)


SC2015 Tutorial

How to Analyze the Performance of Parallel Codes 101

A case study with Open|SpeedShop

Section 10

Hands-On Exercises





Let's Get Our Hands Dirty

- **Setup**
  - Demo cluster in the room
  - O|SS pre-installed
  - Exercises in central location
- **More information in hand-outs**
- **Dedicated exercises**
  - Addressing some of the questions in this tutorial
- **But: FEEL FREE TO EXPERIMENT YOURSELF**
- **Suggestion**
  - Let's use the break to make sure things are setup
  - Let's dive into hands-on work after the break