How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.

Tutorial @ SC ‘11

Donald Frederick LLNL
Motivation

• Performance Analysis is becoming more important
  – Complex architectures
  – Complex applications
  – Mapping applications onto architectures

• 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?
Outline

• Basics on Open|SpeedShop
  – Introduction into one possible tool solution
  – Basic usage instructions
  – Pointers to additional documentation

• Provide you with the ability to ... 
  – Run these experiments on your own code
  – Provide starting point for performance optimizations

• Some useful pointers
  – [http://www.openspeedshop.org][1]
    • Has SC 10 tutorial, quick start guide, OSS download, etc.,
  – [http://www.openspeedshop.org/forums][2]
    • Has question/answer on OpenSpeedShop topics

[1]: http://www.openspeedshop.org
[2]: http://www.openspeedshop.org/forums
OSS Development

- Jim Galarowicz, Krell
- Don Maghrak, Krell

- Tri-Lab ASC CCE Management
  - Martin Schulz, LLNL
  - David Montoya, LANL
  - Mahesh Rajan, SNLs

- Larger team
  - William Hachfeld and Dave Whitney, Krell
  - Dane Gardner, LANL
  - Scott Cranford and Joseph Kenny, SNLs
  - Chris Chambreau and Matt Legendre, LLNL
  - Dyninst group (Bart Miller, UW & Jeff Hollingsworth, UMD)
  - Phil Roth, ORNL
  - Ciera Jaspan, CMU
Why Open | SpeedShop?

• Various alternatives – vendor tools, other open source tools
  – OSS - Powerful, relatively easy to use
  – OSS - Available for both Blue Gene and Cray XT
  – Open source
What to Look For: Sequential Runs

Step 1: Identify computational intensive parts
- Where am I spending my time?
  - Modules/Libraries
  - Statements
  - Functions
- Is the time spent in the computational kernel?
- 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: 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
  – Long blocking times waiting for data
  – Low message rates
  – Limited network bandwidth
  – Global collective communication operations
Performance Tool Terminology

- **Statistical Sampling**
  - Periodically interrupt execution and record location
  - Report statistical distribution across all reported locations
  - Data typically aggregated over time
  - Most common metric is time, but other metrics possible
  - Useful to get an overview
  - Low and uniform overhead

- **Event Tracing**
  - Gather and store individual application events
  - Examples: function invocations, MPI messages, I/O calls
  - Events are typically time stamped
  - Provides detailed per event information
  - Can lead to huge data volumes
  - Higher overhead, potentially in bursts
Section 2
Introduction into Open|SpeedShop

Tutorial @ SC ‘11
November, 2011

How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.
Experiment Workflow

Open|SpeedShop Workflow

Application

“Experiment”

Consists of one or more data “Collectors”

Process Management Panel

Results

Results can be displayed using several “Views”

Run

Stored in SQL database
High-level Architecture

Open SpeedShop™

GUI
CLI
pyOISS

Code Instrumentation
Open Source Software

AMD, Intel, PowerPC based clusters/SSI using Linux

Open Source: PAPI, libmonitor, sqlite, Dyninst, MRNet, libdwarf, libelf, python, etc..
Basic Interface Usage

• Step 1
  – Gather data from command line
  – Example: osspcsamp "<application>"
  – Create database

• Step 2
  – Analyze data in GUI
  – Simple graphics
  – Relate data to source
  – openss -f <db file>
Performance Experiments

• Concept of an Experiment
  – What to measure and what to analyze?
  – Experiment type is chosen by user
  – Any experiment can be applied to any application

• Consists of Collectors and Views
  – Collectors define specific performance data sources
    • Program counter samples
    • Call stack samples
    • Hardware counters
    • Tracing of library routines
  – Views specify performance data aggregation and presentation
  – Multiple collectors per experiment possible
Sampling Experiments

• PC Sampling (pcsamp)
  – Record PC in user defined time intervals
  – 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, whom is calling who

• Hardware Counters (hwc, hwctime, hwcsamp)
  – Access to data like cache and TLB misses
  – hwc, hwctime:
    • Sample a HWC event based on an event threshold
    • Default event is PAPI_TOT_CYC overflows
  – hwcsamp:
    • Sample up to six events based on a sample time (hwcsamp)
    • Default events are PAPI_FP_OPS and PAPI_TOT_CYC
Tracing Experiments

• Input/Output Tracing (io, iot)
  – Record invocation of all POSIX I/O events
  – Provides aggregate and individual timings
  – Provide argument information for each call (iot)

• MPI Tracing (mpi, mpit, mpiotf)
  – Record invocation of all MPI routines
  – Provides aggregate and individual timings
  – Provide argument information 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
Running a First Experiment

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
   – 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"

[openss]: pcsamp experiment using the pcsamp experiment default sampling rate: "100".
[offline pcsamp experiment using the command:
"mpirun -np 2 /opt/OSS-mrnet/bin/ossrun "./smg2000 -n 65 65 65" pcsamp"

[openss]: Using OPENSS_PREFIX installed in /opt/OSS-mrnet

[openss]: Setting up offline raw data directory in /tmp/jeg/offline-oss

[openss]: Running

Running with these driver parameters:
(nx, ny, nz) = (65, 65, 65)
(Px, Py, Pz) = (2, 1, 1)
(bx, by, bz) = (1, 1, 1)
(cx, cy, cz) = (1.000000, 1.000000, 1.000000)
(n_pre, n_post) = (1, 1)
dim = 3
solver ID = 0

=============================================
Struct Interface:
=============================================
Struct Interface:
wall clock time = 0.049847 seconds
cpu clock time = 0.050000 seconds
```
Example Run with Output

============================================
Setup phase times:
SMG Setup:
  wall clock time = 0.635208 seconds
  cpu clock time = 0.630000 seconds
============================================
Solve phase times:
============================================
SMG Solve:
  wall clock time = 3.987212 seconds
  cpu clock time = 3.970000 seconds
Iterations = 7
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 ...

osspcsamp “mpirun–np 2 smg2000 –n 65 65 65”
Example Run with Output

```
• osspcsamp “mpirun –np 2 smg2000 –n 65 65 65”
```

```bash
[openss]: Restoring and displaying default view for:
/home/jeg/DEMS/demos/mpi/openmpi-1.4.2/smg2000/test/smg2000-pcsamp-1.openss

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

<table>
<thead>
<tr>
<th>Exclusive CPU time in seconds.</th>
<th>% of CPU Time</th>
<th>Function (defining location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6300000000</td>
<td>43.060498221</td>
<td>hypre_SMGResidual (smg2000: smg_residual.c,152)</td>
</tr>
<tr>
<td>2.8600000000</td>
<td>33.926453144</td>
<td>hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)</td>
</tr>
<tr>
<td>0.2800000000</td>
<td>3.321470937</td>
<td>hypre_SemiRestrict (smg2000: semi_restrict.c,125)</td>
</tr>
<tr>
<td>0.2100000000</td>
<td>2.491103203</td>
<td>hypre_SemiInterp (smg2000: semi_interp.c,126)</td>
</tr>
<tr>
<td>0.1500000000</td>
<td>1.779359431</td>
<td>opal_progress (libopen-pal.so.0.0.0)</td>
</tr>
<tr>
<td>0.1000000000</td>
<td>1.186239620</td>
<td>mca_btl_sm_component_progress (libmpi.so.0.0.2)</td>
</tr>
<tr>
<td>0.0900000000</td>
<td>1.067615658</td>
<td>hypre_SMGAxpy (smg2000: smg_axpy.c,27)</td>
</tr>
<tr>
<td>0.0800000000</td>
<td>0.948991696</td>
<td>ompi_generic_simple_pack (libmpi.so.0.0.2)</td>
</tr>
<tr>
<td>0.0700000000</td>
<td>0.830367734</td>
<td>__GI_memcpy (libc-2.10.2.so)</td>
</tr>
<tr>
<td>0.0700000000</td>
<td>0.830367734</td>
<td>hypre_StructVectorSetConstantValues (smg2000: struct_vector.c,537)</td>
</tr>
<tr>
<td>0.0600000000</td>
<td>0.711743772</td>
<td>hypre_SMG3BuildRAPSym (smg2000: smg3_setup_rap.c,233)</td>
</tr>
</tbody>
</table>

❖ View with GUI: openss –f smg2000-pcsamp-1.openss
How to use OpenSpeedShop to Analyze the Performance of Parallel Codes?

Performance Data Default view: by Function (Data is sum from all processes and threads)
Select “Functions”, click D-icon

Graphical Representation

Toolbar to switch Views
Statement Report Output View

Performance Data View Choice: Statements
Select "statements, click D-icon"

Statement in Program that took the most time
How to use Open|SpeedShop to Analyze the Performance of Parallel Codes?

Double click to open source window

Use window controls to split/arrange windows

Selected performance data point
Section Summary

- 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 creates default view 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
  - Need a shared file system across all nodes of a cluster
- Start with pcsamp for overview of performance
- Then “home” into performance issues with other experiments, if necessary
Section 3
Sampling Experiments

Tutorial @ SC ’11
November, 2011

How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.
Sampling Experiments in OSS

- **PC Sampling**
  - Approximates CPU Time for Line and Function
  - No Call Stacks
  - Convenience Script: osspcsamp

- **User Time**
  - Inclusive vs. Exclusive CPU Time
  - Includes Call Stacks
  - Convenience Script: ossusertime

- **HW Counters**
  - Samples Hardware Counter Overflows (hwc, hwctime)
  - Periodically samples up to six (6) Hardware Counter Events (hwcsamp)
  - Convenience Scripts: osshwc, osshwctime, osshwcsamp
Running Program Counter Sampling

Offline pcsamp experiment on smg2000 application

Option 1: Convenience script basic syntax

osspcsamp “smg2000 –n 50 50 50”
osspcsamp “smg2000 –n 50 50 50” high

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

Option 2: Explicit openss syntax

openss –offline –f “smg2000 –n 50 50 50” pcsamp

Recommendation: compile code with –g to get statements!
Viewing Flat Profiles

How to use Open|SpeedShop to Analyze the Performance of Parallel Codes?

Performance Data
Default view: by Function
(Data is sum from all processes and threads)
Select “Functions”, click D-icon
**Identifying Critical Regions**

- Profiles show computationally intensive code regions
  - First views: Time spent per functions or per statements
    - Default view reports statistics based on functions
    - For statements, choose Statements View option

- 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
    - Linked Object View option
  - Correct/expected modules?
  - Impact of support and runtime libraries
Adding Context (usertime experiment)

• 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 (shows caller/callee relationships)
  – Hot call paths
    • Paths through application that take most time
The Usertime Experiment

• Provides inclusive/exclusive time
  – Time spent inside a function only (exclusive)
  – Time spent inside a function and its children (inclusive)

• Similar to the pcsamp experiment
  – Collect program counter information
  – But also: collect call stack information at every sample

• Tradeoffs
  – Obtain additional context information
  – Have higher overhead/lower sampling rate
Running Usertime Experiments

Offline usertime experiment on smg2000 application

Option 1: Convenience script basic syntax

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)

Option 2: Explicit openss syntax:

openss –offline –f “smg2000 –n 50 50 50” usertime

Recommendation: compile code with –g to get statements!
Stack Trace Views: Hot Call Path

Hot Call Path
Click HC-icon
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: Thresholding
  - User selects counter
  - Run until a fixed number of events have been reached
  - Take PC sample at that location
  - Reset number of events
  - Ideal number of events (threshold) depends on application
- Basic model 2: Sampling
- Three versions
  - HWC = flat hardware counter profile
  - HWCTime = profiles with context / stacktraces
  - HWCSamp = PC sampling with multiple HWCs
Running hwc and hwctime

Example: Offline hwc[time] experiment on SMG2000
Option 1: Basic Syntax
oshwc[time] “smg2000” <counter> <threshold>

Option 2: Explicit Syntax
setenv OPENSS_HWC_EVENT <counter>
setenv OPENSS_HWC_TRHESHOLD <threshold>
openss –offline –f “smg2000” hwc[time]

Available counter
– Any counter reported by “papi_avail –a” and not derived
– Native counters listed in the PAPI documentation
# Examples of Typical Counters

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

Note: Threshold indications are just rough guidance and depend on the application.
Note: Not all counters exist on all platforms (check with papi_avail)
Running hwcsamp

Example: Offline hwcsamp experiment on SMG2000

Option 1: Basic Syntax
osshwcsamp “smg2000” <event list> <sampling rate>

Option 2: Explicit Syntax
setenv OPENSS_HWCSAMP_EVENTS <event list>
setenv OPENSS_HWCSAMP_RATE <sampling rate>
openss –offline –f “smg2000” hwcsamp

Available counter

- Any counters reported by “papi_avail”
- Native counters listed in the PAPI documentation
  - Native counters also reported by “papi_native_avail”
Viewing hwcsamp Data

- Exclusive Time
- Multiple HWC Event Counts
### Viewing hwcsamp data in CLI

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

<table>
<thead>
<tr>
<th>Exclusive CPU time in seconds.</th>
<th>% of CPU Time</th>
<th>PAPI_TOT_CYC</th>
<th>PAPI_FP_OPS</th>
<th>Function (defining location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.920000000</td>
<td>44.697833523</td>
<td>11772604888</td>
<td>1198486900</td>
<td>hypre_SMGResidual (smg2000: smg_residual.c,152)</td>
</tr>
<tr>
<td>2.510000000</td>
<td>28.620296465</td>
<td>7478131309</td>
<td>812850606</td>
<td>hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)</td>
</tr>
<tr>
<td>0.310000000</td>
<td>3.534777651</td>
<td>915610917</td>
<td>48863259</td>
<td>opal_progress (libopen-pal.so.0.0.0)</td>
</tr>
<tr>
<td>0.300000000</td>
<td>3.420752566</td>
<td>910260309</td>
<td>100529525</td>
<td>hypre_SemiRestrict (smg2000: semi_restrict.c,125)</td>
</tr>
<tr>
<td>0.290000000</td>
<td>3.306727480</td>
<td>874155835</td>
<td>48509938</td>
<td>mca_btl_sm_component_progress (libmpi.so.0.0.2)</td>
</tr>
</tbody>
</table>
```

### How to use Open|SpeedShop to Analyze the Performance of Parallel Codes?

The restored experiment identifier is `-x 1`. The header for `expview` indicates the start of a table showing various performance metrics for different functions. The table includes columns for exclusive CPU time in seconds, percentage of CPU time, PAPI_TOT_CYC, PAPI_FP_OPS, and the function name along with its defining location for each operation.
Viewing hwcsamp data in CLI

```bash
openssl>> expview -v linkedobjects
```

<table>
<thead>
<tr>
<th>Exclusive CPU time</th>
<th>% of CPU Time</th>
<th>PAPI_TOT_CYC</th>
<th>PAPI_FP_OPS</th>
<th>LinkedObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.710000000</td>
<td>87.315968290</td>
<td>22748513124</td>
<td>2396367480</td>
<td>smg2000</td>
</tr>
<tr>
<td>0.610000000</td>
<td>6.908267271</td>
<td>1789631493</td>
<td>126423208</td>
<td>libmpi.so.0.0.2</td>
</tr>
<tr>
<td>0.310000000</td>
<td>3.510758777</td>
<td>915610917</td>
<td>48863259</td>
<td>libopen-pal.so.0.0.0</td>
</tr>
<tr>
<td>0.200000000</td>
<td>2.265005663</td>
<td>521249939</td>
<td>46127342</td>
<td>libc-2.10.2.so</td>
</tr>
<tr>
<td>8.830000000</td>
<td>100.000000000</td>
<td>25975005473</td>
<td>2617781289</td>
<td>Report Summary</td>
</tr>
</tbody>
</table>

```bash
openssl>> expview -m flops
```

```
Mflops  Function (defining location)
478.639000000  hypre_ExchangeLocalData (smg2000: communication.c,708)
456.405900000  hypre_StructAxpy (smg2000: struct_axpy.c,25)
452.758600000  mca_pml_ob1_isend (libmpi.so.0.0.2)
447.734300000  MPI_Irecv (libmpi.so.0.0.2)
```
Section 4
I/O Tracing

Tutorial @ SC ‘11
November, 2011
How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.
Need for Understanding I/O

• I/O could be significant percentage of execution time dependent upon:
  – Checkpoint, analysis output, and 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 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
Running I/O Experiments

Offline io/iot experiment on smg2000 application

Option 1: Convenience script basic syntax

ossio[t] “smg2000 –n 50 50 50” [ default | <list of I/O func>]

– Parameters

I/O Function list to trace (default is all)

Option 2: Explicit openss syntax:

setenv OPENSS_IO_TRACED read,write,dup (only needed to change default)

openss –offline –f “smg2000 –n 50 50 50” io

setenv OPENSS_IOT_TRACED read,write,dup (only needed to change default)

openss –offline –f “smg2000 –n 50 50 50” iot
Additional I/O analysis with OSS

• 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.
### Additional I/O analysis with O\|SS

openss\>>expview

<table>
<thead>
<tr>
<th>I/O Call Time(ms)</th>
<th>% of Total Time</th>
<th>Number of Calls</th>
<th>Function (defining location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.906473000</td>
<td>96.239580457</td>
<td>2</td>
<td>__libc_read (libpthread-2.10.2.so)</td>
</tr>
<tr>
<td>0.394533000</td>
<td>2.120444958</td>
<td>36</td>
<td>__libc_write (libpthread-2.10.2.so)</td>
</tr>
<tr>
<td>0.265676000</td>
<td>1.42784079</td>
<td>72</td>
<td>llseek (libpthread-2.10.2.so)</td>
</tr>
<tr>
<td>0.024794000</td>
<td>0.078823434</td>
<td>2</td>
<td>__libc_close (libpthread-2.10.2.so)</td>
</tr>
</tbody>
</table>

openss\>>expview -vtrace

<table>
<thead>
<tr>
<th>Start Time</th>
<th>I/O Call Time(ms)</th>
<th>% of Total Time</th>
<th>Function Dependent</th>
<th>File/Path Name</th>
<th>Call Stack Function (defining location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011/04/22 15:28:49</td>
<td>0.009847000</td>
<td>0.052923384</td>
<td>10</td>
<td>input</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;____libc_open (libpthread-2.10.2.so)</td>
</tr>
<tr>
<td>2011/04/22 15:28:49</td>
<td>0.003003000</td>
<td>0.016139832</td>
<td>0</td>
<td>/home/jeg/DEMOS/demos/mpi/openmpi-1.4.2/sweep3d/input</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;llseek (libpthread-2.10.2.so)</td>
</tr>
<tr>
<td>2011/04/22 15:28:49</td>
<td>17.902981000</td>
<td>96.220812461</td>
<td>53</td>
<td>/home/jeg/DEMOS/demos/mpi/openmpi-1.4.2/sweep3d/input</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;__libc_read (libpthread-2.10.2.so)</td>
</tr>
<tr>
<td>2011/04/22 15:28:49</td>
<td>0.013479000</td>
<td>0.072443820</td>
<td>0</td>
<td>/home/jeg/DEMOS/demos/mpi/openmpi-1.4.2/sweep3d/input</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;__libc_close (libpthread-2.10.2.so)</td>
</tr>
<tr>
<td>2011/04/22 15:28:49</td>
<td>0.004819000</td>
<td>0.025900050</td>
<td>10</td>
<td>input</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;____libc_open (libpthread-2.10.2.so)</td>
</tr>
<tr>
<td>2011/04/22 15:28:49</td>
<td>0.002095000</td>
<td>0.011259723</td>
<td>0</td>
<td>/home/jeg/DEMOS/demos/mpi/openmpi-1.4.2/sweep3d/input</td>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;llseek (libpthread-2.10.2.so)</td>
</tr>
</tbody>
</table>

How to use Open\|SpeedShop to analyze the Performance of Parallel Codes?
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 5
How to Detect Bottlenecks in Parallel Codes?

Tutorial @ SC ‘11
November, 2011

How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.
Integration with MPI

- O|SS has been tested with a variety of MPIs
  - Including: Open MPI, MVAPICH, and MPICH2 (Intel, Cray)
- Running O|SS experiments on MPI codes
  - Add MPI starter/driver phrase as part of executable name, just as you would when running outside of the O|SS tool
  - ossmpi “mpirun –np 128 sweep3d.mpi”
  - osspcsamp “mpirun –np 32 sweep3d.mpi”
  - ossio “srun –N 4 –n 16 sweep3d.mpi”
  - openss –offline –f “mpirun –np 128 sweep3d.mpi” hwctime
  - openss –online –f “srun –N 8 –n 128 sweep3d.mpi” usertime
Parallel Result Analysis

• Default views
  – Values aggregated across all ranks
  – Manually include/exclude individual ranks/processes/threads

• Comparing ranks
  – Use the Customize Stats Panel View
  – Create compare columns for process groups or individual ranks

• Load Balance View
  – Calculates min, max, average across ranks, processes, or threads

• Comparative Analysis View (finding outliers)
  – Uses “cluster analysis” algorithm to group similar performing ranks, processes, or threads
Running MPI Experiments

Offline mpi/mpit/mpiotf experiment on smg2000 appl.

Option 1: Convenience script basic syntax

ossmpi[t] “srn –N 4 –n 32 smg2000 –n 50 50 50” [default | <list MPI func> | mpi category]

– Parameters

MPI Function list to trace (default is all)

Option 2: Explicit openss syntax:

setenv OPENSS_MPI_TRACED collective_com
setenv OPENSS_MPIT_TRACED MPI_Allgather,MPI_Send
How to use OpenSpeedShop to analyze the Performance of Parallel Codes

Information Icon Displays Experiment Metadata

Aggregated Results

Table:

<table>
<thead>
<tr>
<th>Minimum MPI Call Time (ms)</th>
<th>Maximum MPI Call Time (ms)</th>
<th>Average Time (ms)</th>
<th>Number of Calls</th>
<th>Function (defining location)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PMPI_Init (libmonitor.so.0.0.0: pmpi.c,94)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PMPI_Finalize (libmonitor.so.0.0.0: pmpi.c,223)</td>
</tr>
<tr>
<td>-555.306000</td>
<td>1276.275000</td>
<td>755.289027</td>
<td>512</td>
<td>MPI_Allgatherv (libmpich.so.1.0: allgatherv.c,73)</td>
</tr>
<tr>
<td>-151.147000</td>
<td>167.504000</td>
<td>163.231094</td>
<td>512</td>
<td>MPI_Allgather (libmpich.so.1.0: allgather.c,70)</td>
</tr>
<tr>
<td>-0.152000</td>
<td>0.474000</td>
<td>0.334205</td>
<td>512</td>
<td>MPI_Barrier (libmpich.so.1.0: barrier.c,56)</td>
</tr>
<tr>
<td>-0.043000</td>
<td>0.212000</td>
<td>0.133098</td>
<td>512</td>
<td>MPI_Allreduce (libmpich.so.1.0: allreduce.c,59)</td>
</tr>
<tr>
<td>-0.031000</td>
<td>2.034000</td>
<td>1.312102</td>
<td>512</td>
<td>MPI_Waitall (libmpich.so.1.0: waitall.c,57)</td>
</tr>
<tr>
<td>-0.013000</td>
<td>10.322000</td>
<td>0.717578</td>
<td>6144</td>
<td>MPI_Isend (libmpich.so.1.0: isend.c,58)</td>
</tr>
<tr>
<td>-0.0000001</td>
<td>611.617000</td>
<td>0.977852</td>
<td>4667648</td>
<td>MPI_Irecv (libmpich.so.1.0: irecv.c,48)</td>
</tr>
</tbody>
</table>
How to use Open|SpeedShop to analyze the performance of parallel codes.

**Unique Call Paths View:** Click C+ Icon

**Unique Call Paths to MPI_Waitall and other MPI functions**
Finding Outliers: Using NPB: LU

In Cluster Analysis results Rank 255 showing up as an outlier.
Load Balance View: Using NPB: LU

Rank 255 has maximum MPI library time value.
Section 6
User Interfaces

Tutorial @ SC ‘11
November, 2011
How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.
CLI Command Overview

- **Experiment creations**
  - expcreate
  - expattach*

- **Experiment control**
  - expgo
  - expwait*
  - expdisable*
  - expenable*

- **Experiment storage**
  - expsave
  - exprestore

- **Result presentation**
  - expview
  - opengui

- **Misc. commands**
  - help
  - list
  - log
  - record
  - playback
  - history
  - quit

*online version only*
CLI Command Examples

- Simple usage to create, run, view data
  - `openss --cli` (use cli to run experiment:3 commands)
    - `expcreate --f "mutatuee 2000" pcsamp` (create an experiment with instrumentation added for the particular collector)
    - `expgo` (runs the experiment gathering data into database)
    - `expview` (displays the default view of the performance data)

- Alternative views of the performance data
  - `openss --cli --f <database file name>`
    - `expview --v statements` (see the statements that took the most time)
    - `expview --v linkedobjects` (see time attributed to the libraries in appl.)
    - `expview --v calltrees, fullstack` (see all the call paths in application)
    - `expview --m loadbalance` (see the min, max, average across ranks/threads)
    - `list --v metrics` (display the optional performance data metrics)
    - `expview --m <metric from above>` (view the metric specified)
Section 7
Comparing Experiment Results

Tutorial @ SC ‘11
November, 2011

How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.
Comparing Performance Data (1)

• GUI Custom Compare Panel (CC icon)
  – See users guide or tutorials for details
• Convenience Script: osscompare
  – Compares Open|SpeedShop databases to each other
  – Compare up to 8 at one time
    • If output file name is too long use:
      • export OPENSS_USE_INTERNAL_NAMING=1
    • Causes shortened output file names to be created
  – Produces side-by-side comparison listing
  – Optionally create "csv" output for input into spreadsheet (Excel,..)
    • export OPENSS_CREATE_CSV=1
• osscompare syntax:
  – osscompare “db1.openss,db2.openss,...” [options]
  – osscompare man page has more details
Comparing Performance Data (2)

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

openss]: Legend: -c 2 represents smg2000-pcsamp.openss
[openss]: Legend: -c 4 represents smg2000-pcsamp-1.openss
-c 2, Exclusive CPU  -c 4, Exclusive CPU  Function (defining location)

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.870000000</td>
<td>hypre_SMGResidual (smg2000: smg_residual.c,152)</td>
</tr>
<tr>
<td>2.610000000</td>
<td>hypre_CyclicReduction (smg2000: cyclic_reduction.c,757)</td>
</tr>
<tr>
<td>2.030000000</td>
<td>opal_progress (libopen-pal.so.0.0.0)</td>
</tr>
<tr>
<td>1.330000000</td>
<td>mca_btl_sm_component_progress (libmpi.so.0.0.2: topo Unity_component.c,0)</td>
</tr>
<tr>
<td>0.280000000</td>
<td>hypre_SemiInterp (smg2000: semi_interp.c,126)</td>
</tr>
<tr>
<td>0.280000000</td>
<td>mca_pml_ob1_progress (libmpi.so.0.0.2: topo Unity_component.c,0)</td>
</tr>
</tbody>
</table>
Section 8
Static Binary Support

Tutorial @ SC ‘11
November, 2011
How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.
Workflow Differences for Static Binary

- When shared library support is limited
  - Normal manner of running experiments doesn’t work
  - Need to link our collectors into the static executable
- Current method (static) of support on Cray and BG/P
- Must relink application to add in OSS collector code.
  - `osslink` command is provided to help
- Have the OpenSpeedShop target NOT host environment module loaded when relinking
Open|SpeedShop & Static Executables

- **osslink**: A script to help with linking in our collectors
  - `osslink` is a script that hides a lot of the link details
  - Calls to it are usually embedded inside application makefiles
  - Can also be used to compile and link applications
  - Sorts the experiment specific library and collector specification
  - Sorts out some platform differences to do the correct link

- **Find Makefile target to add entry for relink with collector**
  - The user generally needs find the target that creates the actual static executable and create a collector target that links in the selected collector as shown in the example.

- **Example follows ->**
Re-linking application using osslink

- Example modification for smg2000 application

```bash
smg2000: smg2000.o
    @echo "Linking" $@ "..."
    $(CC) -o smg2000 smg2000.o $(LFLAGS)

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

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

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

smg2000-io: smg2000.o
    @echo "Linking" $@ "..."
    osslink -u open -v -c io $(CC) -o smg2000-io smg2000.o $(LFLAGS)

smg2000-iot: smg2000.o
    @echo "Linking" $@ "..."
    osslink -u open -v -c iot $(CC) -o smg2000-iot smg2000.o $(LFLAGS)

smg2000-mpi: smg2000.o
    @echo "Linking" $@ "..."
    osslink -v -c mpi $(CC) -o smg2000-mpi smg2000.o $(LFLAGS)
```
Workflow Differences for Static Binary

- Running the re-linked executable causes the application to write the raw data files to the location specified by OPENSS_RAWDATA_DIR
- Converting raw data to an OpenSpeedShop database
  - Normally the convenience scripts or openss tool will do this under the hood
  - ossutil command will create the database file.
  - If raw data files are in /home/jgalaro/smg2000/test/raw
    - Then ossutil /home/jgalaro/smg2000/test/raw  # creates a database file
    - Symbol table information is read and mapped to the raw data application addresses to map source to the performance data.
- The ossutil step may be added to the batch script to eliminate an additional step
- Now normal OpenSpeedShop GUI and CLI viewing of the performance data is possible
Workflow Differences for Static Binary

• **Now run executable:** (note –bb argument to aprun)

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

setenv OPENSS_RAWDATA_DIR /home/jgalaro/smg2000/test/raw
setenv OPENSS_DB_DIR /home/jgalaro/sm2000/test/

cd /home/jgalaro/smg2000/test
# needs –bb to have the original executable available when doing ossutil
aprun -bb -n 16 /home/jgalaro/smg2000/test/sm2000-pcsamp

# creates a X.0.openss database file, may need to load module pointing to openspeedshop here
ossutil /home/jgalaro/smg2000/test/raw
```
Open|SpeedShop on Cray/BG Systems

• Changes to shared library support recently
  – Dynamic shared library support now available in newer O/S
  – We are working on providing shared support in addition to static binary support
  – This will allow the normal Open|SpeedShop workflow to be used, for example:
    • osspcsamp “how you run the executable dynamically”

• Hoping to have this shared support in Open|SpeedShop in the next 3 months.
Conclusions

Tutorial @ SC ‘11
November, 2011
How to use Open|SpeedShop to Analyze the Performance of Parallel Codes.
Questions vs. Experiments

• Where do I spend my time?
  – Flat profiles (pcsamp)
  – Getting inclusive/exclusive timings with callstacks (usertime)
  – Identifying hot callpaths (usertime + HC 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)
Key Environment Variables

- Important runtime environment variables (module file)
  - `OPENSS_RAWDATA_DIR`
    - Shared (across nodes) file system dir where raw data files are written
  - `OPENSS_DB_DIR`
    - Need a file system with locking enabled: directory path for database
    - lustre file system may not have locking enabled
  - `OPENSS_PLUGIN_PATH`
    - Path to directory where plugins are stored
  - `OPENSS_MPI_IMPLEMENTATION` (if multiple)
    - If Open|SpeedShop was built with multiple MPI implementations, this points openss at the one you are using in your application
    - Also, only required if using the mpi, mpit, or mpiotf experiments
  - `LD_LIBRARY_PATH`, `PATH`
    - Linux path variables

- Example
  ```bash
  export OPENSS_PREFIX=/home/skg/local
  export OPENSS_MPI_IMPLEMENTATION=openmpi
  export OPENSS_PLUGIN_PATH=$OPENSS_PREFIX/lib64/openspeedshop
  export LD_LIBRARY_PATH=$OPENSS_PREFIX/lib64:$LD_LIBRARY_PATH
  export PATH=$OPENSS_PREFIX/bin:$PATH
  ```
Open|SpeedShop Documentation

- Open|SpeedShop User Guide Documentation
  - [http://www.openspeedshop.org/docs/user_guide/](http://www.openspeedshop.org/docs/user_guide/)
  - .../share/doc/packages/OpenSpeedShop/users_guide
- Python Scripting API Documentation
  - .../share/doc/packages/OpenSpeedShop/pyscripting_doc
- Command Line Interface Documentation
  - [http://www.openspeedshop.org/docs/user_guide/](http://www.openspeedshop.org/docs/user_guide/)
  - .../share/doc/packages/OpenSpeedShop/users_guide
- Man pages: OpenSpeedShop, ossppcsamp, ossmpi, ...
- Quick start guide downloadable from web site
  - [http://www.openspeedshop.org](http://www.openspeedshop.org)
  - Click on “Download Quick Start Guide” button
Availability and Contact

- Current version: 2.0.1 about to be released

- Open|SpeedShop Website
  - [http://www.openspeedshop.org/](http://www.openspeedshop.org/)

- Open|SpeedShop Forum
  - [http://www.openspeedshop.org/forums/](http://www.openspeedshop.org/forums/)

- Download options:
  - Package with install script
  - Source for tool and base libraries

- Feedback
  - Bug tracking available from website
  - Contact information on website
  - Feel free to OSS team directly
How to Learn More about OSS

This short tutorial gives a hint at the full power of OSS

• Sign up for SC ‘11 Monday 1/2-day OSS Tutorial!
Acknowledgements

• Jim Galarowicz, Krell
• Mahesh Rajan, SNL

• Larger team
  – David Montoya, LANL
  – Donald Maghrak, Krell
  – Martin Schulz, LLNL
  – William Hachfeld and Dave Whitney, Krell
  – Dane Gardner, Argo Navis/Krell
  – Chris Chambreau and Matt Legendre, LLNL
  – Dyninst group (Bart Miller, UW & Jeff Hollingsworth, UMD)
  – Phil Roth, ORNL
  – Ciera Jaspan, CMU