Understanding Performance of Parallel Codes
Using Open|SpeedShop
SEA@UCAR – Boulder, CO - April 3, 2013

Jim Galarowicz : Krell Institute
Martin Schulz : LLNL
Open SpeedShop Overview

❖ Open Source Performance Analysis Tool Framework
  ➢ Most common performance analysis steps all in one tool
  ➢ Gathers and displays several types of performance information
    • Profiling options to get initial overview
    • Detailed analysis views to drill down to locate bottlenecks
    • Event tracing options to capture performance details
  ➢ Various analysis options for comparisons, load balance, ...

❖ Flexible and Easy to use
  ➢ User access through GUI, Command Line, Python Scripting, and convenience scripts.
  ➢ Gather performance data from unmodified application binaries

❖ Supports a wide range of systems
  ➢ Extensively used and tested on a variety of Linux clusters
    • In preparation for SEA adapted to IBM’s cluster software
  ➢ Cray XT/XE/XK and Blue Gene P/Q support
OSS Team and History

- Jim Galarowicz, Krell
- Martin Schulz, LLNL
- Larger team
  - Don Maghrak, Krell
  - William Hachfeld, Dave Whitney, Krell
  - Dane Gardner, Argo Navis/Krell
  - Matt Legendre, LLNL
  - Jennifer Green, Philip Romero, LANL
  - David Montoya, David Gunther, Michael Mason, LANL
  - Mahesh Rajan, Anthony Agelastos, SNLs
  - Dyninst group (Bart Miller, UW & Jeff Hollingsworth, UMD)
  - Phil Roth, Michael Brim, ORNL

- Project history:
  - Started at SGI in 2004
  - 6-8 developers for two years at SGI
  - Krell and Tri-labs agreement and funding since late 2006
  - Office of Science and DOE STTR and SBIR funding over the time period since 2006
  - Tool available as open source, support through maintenance contracts with Krell
Open SpeedShop Workflow

Application

“Experiment”

Consists of one or more data “Collectors”

Results can be displayed using several “Views”

convenience Scripts
osspcsamp, etc. or
osslink -c pcsamp
ossutil <raw data>

Run

Results

Stored in SQL database

SEA@UCAR - Boulder, CO
A First Simple Example

- Using PC Sampling experiments
  - Good initial experiment that shows where time is spent

>> osspcsamp "mpirun –np 2 smg2000 –n 65 65 65"

MPI Application

O|SS

Post-mortem
Installation / Usage on Yellowstone

- **Installed in central location**
  - Use:
    - module use /glade/u/home/galaro/privatemodules
    - module load openss
  - Adds GUI and all scripts into the path

- **Starting an MPI job through LSF**
  - Using mpirun.lsf <binary> from within a batch script

- **A few things to consider**
  - Location of raw data files
    - Environment variable: OPENSS_RAWDATA_DIR
    - On yellowstone by default set to: /glade/scratch/${USER}
  - Additional environment variables or arguments to convenience scripts can control each experiment
    - Sampling rates, types of counters, subsets of functions to be traced
    - More on environment variables in the tutorial
Example Run with Output (Startup)

- **osspcsamp “mpirun –np 2 smg2000 –n 65 65 65”**

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 /opt/shared/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)
(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 (App. term.)

- osspcsamp “mpirun –np 2 smg2000 –n 65 65 65”

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 /opt/shared/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 (Results)

**osspcsamp “mpirun –np 2 smg2000 –n 65 65 65”**

[openss]: Restoring and displaying default view for:
/home/jeg/DEMOS/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**
Default Output Report View

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

Toolbar to switch Views

Graphical Representation
Performance Data
View Choice: Statements
Select “statements, click D-icon

Statement in Program that took the most time
Associate Source & Performance Data

Double click to open source window

Use window controls to split/arrange windows

Selected performance data point
Available Experiments I (Profiling)

- **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 (hwcsamp, hwc, hwctime)**
  - Access to data like cache and TLB misses
  - hwcsamp:
    - Sample up to six events based on a sample time (hwcsamp)
    - Default events are PAPI_FP_OPS and PAPI_TOT_CYC
    - Good overview of hardware counter event counts
  - hwc, hwctime:
    - Sample a HWC event based on an event threshold
    - Default event is PAPI_TOT_CYC overflows
Available Experiments II (Tracing)

- 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)
  - Record invocation of all MPI routines
  - Provides aggregate and individual timings
  - Provide argument information for each call (mpit)
  - Optional experiments to create OTF/Vampir files

- Floating Point Exception Tracing (fpe)
  - Triggered by any FPE caused by the application
  - Helps pinpoint numerical problem areas

*More details and examples on all experiments during the OSS tutorial in the coming days*
Future Experiments by End of 2013

- **New Open|SpeedShop experiments under construction**
  - Lightweight I/O experiment (iop)
    - Profile I/O functions by recording individual call paths
      - Rather than every individual event with the event call path, (io and iot).
      - More opportunity for aggregation and smaller database files
    - Map performance information back to the application source code.
  - Memory analysis experiment (mem)
    - Record and track memory consumption information
      - How much memory was used – high water mark
      - Map performance information back to the application source code
  - Threading analysis experiment (thread)
    - Report statistics about pthread wait times
    - Report OpenMP (OMP) blocking times
    - Attribute gathered performance information to proper threads
    - Thread identification improvements
      - Use a simple integer alias for POSIX thread identifier
    - Report synchronization overhead mapped to proper thread
    - Map performance information back to the application source code
Analysis of Parallel Codes

- By default experiment collectors are run on all tasks
  - Automatically detect all processes and threads
  - Gathers and stores per thread data

- Viewing data from parallel codes
  - By default all values aggregated (summed) across all ranks
  - Manually include/exclude individual ranks/processes/threads
  - Ability to compare ranks/threads

- Additional analysis options
  - Load Balance (min, max, average) across parallel executions
    - Across ranks for hybrid openMP/MPI codes
    - Focus on a single rank to see load balance across OpenMP threads
  - Cluster analysis (finding outliers)
    - Automatically creates groups of similar performing ranks or threads
    - Available from the Stats Panel toolbar or context menu
Alternative Interfaces

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

- **Python module**

Experiment Commands
expAttach
expCreate
expDetach
expGo
expView

List Commands
list -v exp

```python
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)
```
### CLI Example: sweep3d I/O Experiment

```
openss –cli –f sweep3d-io.openss

openss>>expview
```

<table>
<thead>
<tr>
<th>I/O Call</th>
<th>Total of Time(ms)</th>
<th>Calls</th>
<th>Number</th>
<th>Function (defining location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>18.01600</td>
<td>36</td>
<td>77.255575</td>
<td>(sweep3d-io)</td>
</tr>
<tr>
<td>open64</td>
<td>2.36400</td>
<td>2</td>
<td>10.137221</td>
<td>(sweep3d-io)</td>
</tr>
<tr>
<td>read</td>
<td>1.30700</td>
<td>2</td>
<td>5.604631</td>
<td>(sweep3d-io)</td>
</tr>
<tr>
<td>__lseek64</td>
<td>1.04000</td>
<td>72</td>
<td>4.459691</td>
<td>(sweep3d-io)</td>
</tr>
<tr>
<td>__close</td>
<td>0.59300</td>
<td>2</td>
<td>2.542882</td>
<td>(sweep3d-io)</td>
</tr>
</tbody>
</table>

```

openss>>expview -vcalltrees,fullstack io1

```
```

<table>
<thead>
<tr>
<th>I/O Call</th>
<th>Total of Time(ms)</th>
<th>Calls</th>
<th>Number</th>
<th>Call Stack Function (defining location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__libc_start_main</td>
<td>5.588000</td>
<td>1</td>
<td>23.962264</td>
<td>(sweep3d-io: libc-start.c,118)</td>
</tr>
<tr>
<td>&gt; @ 226 in generic_start_main</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: libc-start.c,96)</td>
</tr>
<tr>
<td>&gt;&gt; __wrap_main</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io)</td>
</tr>
<tr>
<td>&gt;&gt;&gt; monitor_main</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt; @ 21 in main</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: fmain.c,11)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt; @ 184 in MAIN_</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: driver.f,1)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; @ 66 in inner_auto</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: inner_auto.f,2)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; @ 164 in inner</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: inner.f,2)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; @ 3339 in _gfortran_st_write_done</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: transfer.c,3333)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; @ 3228 in finalize_transfer</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: transfer.c,3142)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; @ 3132 in _gfortran_next_record</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: transfer.c,3100)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; @ 70 in _gfortran_ctrl只不过</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: fbuf.c,143)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt; @ 261 in raw_write</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io: unix.c,250)</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;&gt;write</td>
<td></td>
<td></td>
<td></td>
<td>(sweep3d-io)</td>
</tr>
</tbody>
</table>
Comparing Performance Data

- **Comparisons: basic operation for performance analysis**
  - Compare performance before/after optimization
  - Track performance during code history
  - Compare ranks to each other

- **Open|SpeedShop enables flexible comparisons**
  - Within databases and across multiple databases
  - Within the same experiment and across experiments

- **Convenience Script: osscompare**
  - Compares Open|SpeedShop databases to each other
  - Compare up to 8 at one time
  - Produces side-by-side comparison listing
  - Optionally create "csv" output for input into spreadsheet (Excel,..)
    - export `OPENSS_CREATE_CSV=1`
Example: Comparison Results

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

[openss]: Legend: -c 2 represents smg2000 pcsamp openss
[openss]: Legend: -c 4 represents smg2000 pcsamp-1 openss

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

time in seconds.  

<table>
<thead>
<tr>
<th>Function</th>
<th>time in seconds.</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>hypre SMGResidual (smg2000: smg_residual.c,152)</td>
<td>3.8700000000</td>
<td></td>
</tr>
<tr>
<td>hypre CyclicReduction (smg2000: cyclic_reduction.c,757)</td>
<td>2.6100000000</td>
<td></td>
</tr>
<tr>
<td>opal_progress (libopen-pal.so.0.0.0)</td>
<td>2.0300000000</td>
<td></td>
</tr>
<tr>
<td>mca_btl_sm_component_progress (libmpi.so.0.0.2: topo Unity_component.c,0)</td>
<td>1.3300000000</td>
<td></td>
</tr>
<tr>
<td>hypre SemilInterp (smg2000: semi_interp.c,126)</td>
<td>0.2800000000</td>
<td></td>
</tr>
<tr>
<td>mca_pml_ob1_progress (libmpi.so.0.0.2: topo Unity_component.c,0)</td>
<td>0.2800000000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.0400000000</td>
<td></td>
</tr>
</tbody>
</table>
Scaling Open|SpeedShop

- Open|SpeedShop designed for traditional clusters
  - Tested and works well up to 1,000-10,000 cores
  - Scalability concerns on machines with 100,000+ cores
  - Target: ASC capability machines like LLNL’s Sequoia (20 Pflop/s BG/Q)

- Component Based Tool Framework (CBTF)
  - [http://ft.ornl.gov/doku/cbtfw/start](http://ft.ornl.gov/doku/cbtfw/start)
  - Based on tree based communication infrastructure
  - Porting O|SS on top of CBTF

- Improvements:
  - Direct streaming of performance data to tool without temp. I/O
  - Data will be filtered (reduced or combined) on the fly
  - Emphasis on scalable analysis techniques

- Initial prototype exists, working version: Mid-2013
  - Little changes for users of Open|SpeedShop
  - CBTF can be used to quickly create new tools
  - Additional option: use of CBTF in applications to collect data
Summary

❖ Open Source Performance Analysis Tool Framework
  ➢ Most common performance analysis steps all in one tool
  ➢ Includes a set of experiments:
    • Profiling/Sampling experiments, like pcsamp, to get initial overview
    • Multiple views and analysis options (statement view, comparisons, ...)
    • Event tracing options, like I/O tracing, to capture performance details
  ➢ Special parallel analysis options: load balance & clustering

❖ Flexible and Easy to use
  ➢ User access through multiple interfaces
  ➢ Convenience scripts make it easy to start, just run “oss<exp>”
  ➢ Gather performance data from unmodified application binaries

❖ Supports a wide range of systems
  ➢ Extensively used and tested on a variety of Linux clusters
    • Available on yellowstone
  ➢ Cray XT/XE/XK and Blue Gene P/Q support
Guided tutorial with hands-on exercises
  • Bring your own code or use one of our demos
Covers sequential and parallel codes
  • Basic profiling experiments, incl. hardware counters
  • MPI and I/O experiments
  • GUI and command line access
Access to yellowstone required (preferably with X connection)

Open discussion and “bring your own code clinic”
  • General Q&A
  • Working on advanced features
  • Experimentation on and analysis of your own codes
If people are interested:
  • Help with installing O|SS on other systems beyond yellowstone
  • Introduction into CBTF
Availability and Contact

- **Current version: 2.0.2 update 8**
  - On yellowstone:
    - module use /glade/u/home/galaro/privatemodules
    - module load openss

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

- **Open|SpeedShop Quick Start Guide:**

- **Feedback**
  - Bug tracking available from website
  - Contact information on website
  - [oss-questions@openspeedshop.org](mailto:oss-questions@openspeedshop.org)
  - Open|SpeedShop Forum
    - [http://www.openspeedshop.org/forums/](http://www.openspeedshop.org/forums/)