Cray Performance Tools

Heidi Poxon
Sr. Principal Engineer
Cray Inc.

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publicly announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, URIKA and YARCDATA. The following are trademarks of Cray Inc.: CHAPEL, CLUSTER CONNECT, CLUSTERSTOR, CRAYDOC, CRAYPAT, CRAYPORT, DATAWARP, ECOPHLEX, LIBSCI, NODEKARE, REVEAL. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used on this website are the property of their respective owners.

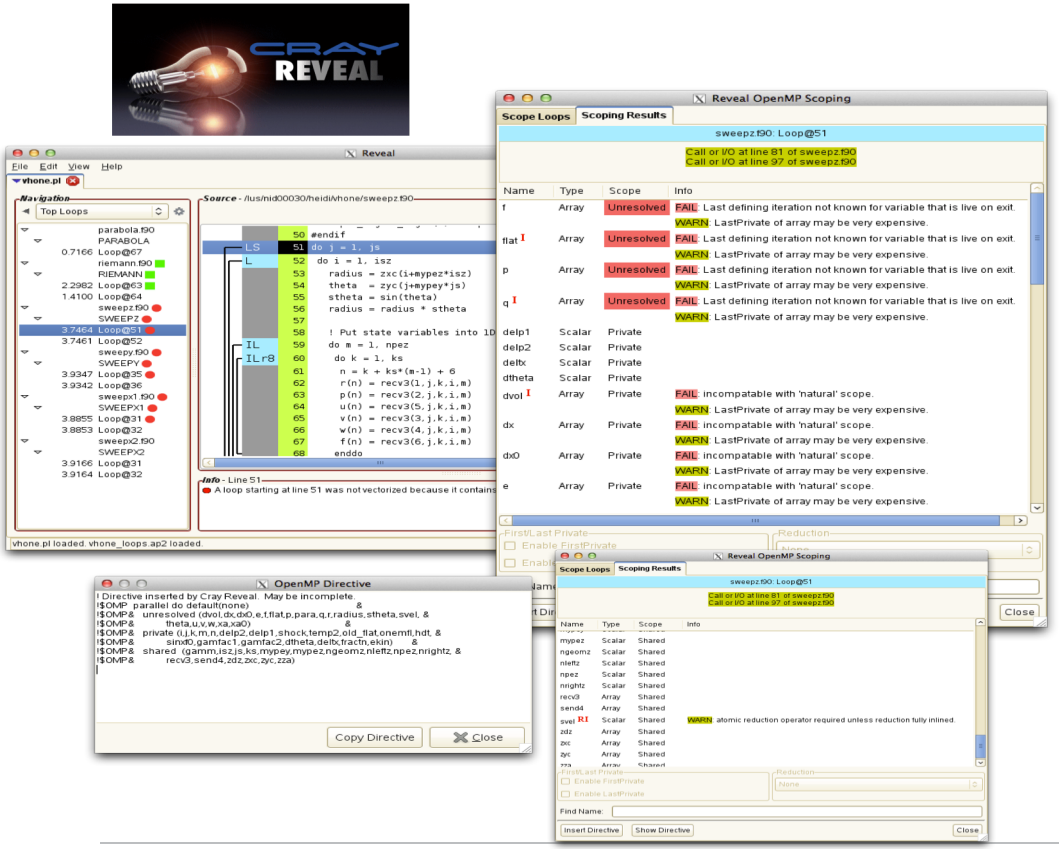
Focus of This Presentation

- **Improve your familiarity with the Cray performance tools**
  - Add to your bag of tricks for application performance tuning
  - Review the mechanics of using Cray performance tools
  - Learn how to identify problem areas and learn which tool to use when

Cray Performance Tools

- Reduce the time investment associated with porting and tuning applications on Cray systems
- Analyze whole-program behavior across many nodes to identify critical performance bottlenecks within a program
- Improve profiling experience by using simple and/or advanced interfaces for a wealth of capability that targets analyzing the largest HPC jobs

Reveal



- Reduce effort associated with adding OpenMP to MPI programs
- Get insight into optimizations performed by the Cray compiler
- Add OpenMP as a first step to parallelize loops that will target GPUs
- Track requests to memory and evaluate the bandwidth contribution of objects within a program for loop tuning

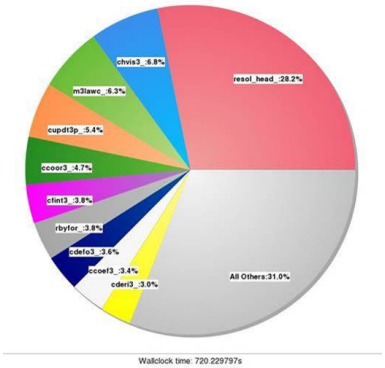
Interfaces Available

- **Simple interface (perftools-lite modes) for convenience**
- **Advanced interface (perftools) for in-depth performance investigation and tuning assistance as well as data collection control**
- **Both offer:**
  - Whole program analysis across many nodes
  - Indication of causes of problems
  - Ability to easily switch between the two interfaces

New!

pat_run

Profile pre-existing, dynamically linked programs



Collect different performance data for same binary

Get basic performance information on ISV codes

What About Different Compilers?

Cray Performance Tools support the following compilers

- **Cray (CCE), Intel, GCC, and Arm Allinea** compilers on Cray XC systems
- **Cray (CCE)** compiler on Cray CS systems

Some Useful Experiments

- **Identify slowest areas and notable bottlenecks of a program**
  - Use `perftools-lite`
  - Good for examining performance characteristics of a program and for scaling analysis
- **Focus on MPI communication**
  - Use `perftools-lite` first to determine if MPI time is dominant or if there is a load imbalance between ranks
  - Use `perftools (pat_build –g mpi)` to collect more detailed MPI-specific information including MPI_SYNC time to detect late arrivers to collectives
  - Good for identifying source of imbalance and scaling analysis at targeted final job size
- **Focus on loop optimization**
  - Use `perftools-lite-loops`
  - Good for vectorizing, parallelizing and cache optimization

Identify Slowest Areas of a Program (perftools-lite)

- `user@login> module load perftools-lite`
- Build program
- Run program
- View report to STDOUT (and to .rpt file in experiment directory)
  - Example data directory: stencil_order+49144-225s/

Consolidated Performance Data

- Available starting with perftools 6.5.0
- **Easily access performance data**
- Unique directory name for each experiment
- Same prefix naming scheme as used with multiple xf files
  - `user@login> pat_report expdir > full_report`
  - `user@login> app2 vhone+73030-20s`
- Example directory:
  - `user@login> ls vhone+73030-20s`
    `ap2-files/  index.ap2  rpt-files/  xf-files/`

Get Additional Information Without Re-running

- **Generate full report**
  - `user@login> pat_report my_data_directory+12s/ > rpt`
- **Generate report with call tree (or by callers)**
  - `user@login> pat_report –O ct+src`
- **Generate report without pruning**
  - `user@login> pat_report –P`
- **Show each MPI rank or each OpenMP thread in report**
  - `user@login> pat_report –s pe=ALL`
  - `user@login> pat_report –s th=ALL`

Example: perftools-lite Job Summary

```
CrayPat/X: Version 7.0.1 Revision 3714888 03/07/18 02:11:13
Experiment:                  lite lite/sample_profile
Number of PEs (MPI ranks):    36
Numbers of PEs per Node:      36
Numbers of Threads per PE:     1
Number of Cores per Socket:   18
Execution start time: Thu Mar 15 11:14:05 2018
System name and speed: nid00030 2.101 GHz (nominal)
Intel Broadwell CPU Family: 6 Model: 79 Stepping: 1

Avg Process Time:                    3.70 secs
High Memory:                     1,801.4 MBytes   50.0 MBytes per PE
Observed CPU clock boost:          117.2 %
Percent cycles stalled:             38.3 %
Vector intensity (packed instr):     2.6 %
Instr per Cycle:                     1.51
I/O Read Rate:                   3.676263 MBytes/sec
I/O Write Rate:                  0.293086 MBytes/sec
```

Example: perftools-lite Top Time Consumers

```
Table 1:  Profile by Function Group and Function (top 10 functions shown)
  Samp% |     Samp |    Imb. |  Imb. |Group
        |          |    Samp | Samp% | Function
        |          |         |       |  PE=HIDE
 100.0% | 55,605.7 |      -- |    -- |Total
|-------------------------------------------------------------------------
|  56.5% | 31,412.8 |      -- |    -- |USER
||------------------------------------------------------------------------
||  19.7% | 10,944.1 |   290.9 |  2.6% |create_boundary$boundary_
||  10.7% |  5,937.8 |   214.2 |  3.5% |get_block$blocks_
||   3.9% |  2,194.4 |     7.6 |  0.3% |create_distrb_balanced$distribution_
||   2.0% |  1,135.5 |   137.5 | 10.8% |impvmixt$vertical_mix_
||   1.9% |  1,064.8 |   124.2 | 10.5% |impvmixt_correct$vertical_mix_
||========================================================================
|  22.5% | 12,513.4 |      -- |    -- |ETC
||------------------------------------------------------------------------
||  20.1% | 11,151.4 | 2,758.6 | 19.9% |__cray_memcpy_KNL
||========================================================================
|  20.7% | 11,503.5 |      -- |    -- |MPI
||------------------------------------------------------------------------
||  11.1% |  6,171.6 | 1,785.4 | 22.5% |MPI_ALLREDUCE
||   7.9% |  4,377.8 | 3,254.2 | 42.7% |mpi_waitall
```

Example: perftools-lite Observations

```
MPI Grid Detection:
    There appears to be point-to-point MPI communication in a 32 X 32
    grid pattern. The 20.7% of the total execution time spent in MPI
    functions might be reduced with a rank order that maximizes
    communication between ranks on the same node. The effect of several
    rank orders is estimated below.

    A file named MPICH_RANK_ORDER.Grid was generated along with this
    report and contains usage instructions and the Hilbert rank order
    from the following table.

    Rank Order    On-Node    On-Node  MPICH_RANK_REORDER_METHOD
                Bytes/PE  Bytes/PE%
                          of Total
                          Bytes/PE

       Hilbert  1.413e+12     81.94%  3
           SMP  1.053e+12     61.04%  1
          Fold  9.405e+11     54.53%  2
    RoundRobin  8.962e+11     51.96%  0
```

Example: perftools-lite Hot Spots by Line

```
Table 3:  Profile by Group, Function, and Line
  Samp% |      Samp |     Imb. |  Imb. | Group
        |           |     Samp | Samp% |  Function
        |           |          |       |   Source
        |           |          |       |    Line
        |           |          |       |     PE=HIDE
 100.0% | 60,665.8 |       -- |    -- | Total
|-------------------------------------------------------------------------
|  94.6% | 57,390.6 |       -- |    -- | USER
||------------------------------------------------------------------------
||  82.1% | 49,835.3 |       -- |    -- | LAMMPS_NS::PairLJCut::compute
|||-----------------------------------------------------------------------
3||  80.7% | 48,970.1 |       -- |    -- | src/Obj_xc30intel/../pair_lj_cut.cpp
||||----------------------------------------------------------------------
4|||   3.9% |  2,359.8 |   100.2 |  4.1% | line.102
4|||   1.0% |    596.2 |    61.8 |  9.5% | line.105
4|||   8.3% |  5,022.4 |   683.6 | 12.1% | line.107
4|||   2.9% |  1,744.2 |   966.8 | 36.0% | line.108
```

Don’t See an Expected Function?

- To make the profile easier to interpret, samples are attributed to a caller that is either a user defined function, or a library function called directly by a user defined function
- To disable this adjustment, and show functions actually sampled, use the ‘`pat_report –P`’ option to disable pruning
- You should be able to see the caller/callee relationship with ‘`pat_report -P -O callers`’

Don’t See an Expected Function? (cont’d)

- **Why don’t I see a particular function in the report?**
- **Cray tools filter out data that may distract you**
  - Use **`pat_report –T`** to see functions that didn’t take much time
- **Still don’t see it?**
  - Check the compiler listing to see if the function was inlined

What is ETC Group in the Report?

- When a function is called that cannot be attributed to a user-defined parent function, it gets placed in ETC
- Try '`pat_report –P`'
- **Note:** pat_report depends on the accuracy of the DWARF issued by the compiler

How Do I See per-Rank or per-Thread Data?

- `$ pat_report –s pe=ALL`
- `$ pat_report –s th=ALL`

Focus on MPI Communication Bottlenecks

- `user@login>` **`module load perftools`**
- **Build program**
  - Remember to add **`–hlist=a`** to build with CCE listing
- **Instrument program, only focusing on MPI**
  - `user@login>` **`pat_build –g mpi ./my_program`**
- **Run instrumented program (my_program+pat)**
- **Create report**
  - `user@login>` **`pat_report my_data_directory+12t/ > my_report`**

Focus on Loop Optimization – Find Top Loops

- **`$ module load perftools-lite-loops`**
- **Build program with CCE**
  - Should see messages from CrayPat during build saying that it created an instrumented executable
  - Remember to add **`–hlist=a`** to build with CCE listing
  - Add **`–hpl=/path_to_program_library/my_program.pl`** if you want to use Reveal
- **Run program**
- **Performance data sent to `STDOUT` and to directory with unique name**

Example Loop Statistics (to STDOUT)

Table 2: Loop Stats by Function

Loop Incl Time / Total	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.] PE=HIDE
8.995914	100	25	0	25	sweepy_.LOOP.1.li.33
8.995604	2500	25	0	25	sweepy_.LOOP.2.li.34
8.894750	50	25	0	25	sweepz_.LOOP.05.li.49
8.894637	1250	25	0	25	sweepz_.LOOP.06.li.50
4.420629	50	25	0	25	sweepx2_.LOOP.1.li.29
4.420536	1250	25	0	25	sweepx2_.LOOP.2.li.30
4.387534	50	25	0	25	sweepx1_.LOOP.1.li.29
4.387457	1250	25	0	25	sweepx1_.LOOP.2.li.30
2.523214	187500	107	0	107	riemann_.LOOP.2.li.63

Documentation

- Release Notes
  - `> module help perftools-base/`*`version_number`*
- User manual “Using the Cray Performance Measurement and Analysis Tools” available at http://pubs.cray.com
- pat_help – interactive help utility on the Cray Performance toolset
- Man pages

Man Pages

- **intro_craypat(1)**
  - Introduces the craypat performance tool
  - Runtime environment variables (enable full trace, etc.)
- **pat_build(1)**
  - Instrument a program for performance analysis
- **pat_help(1)**
  - Interactive online help utility
- **pat_report(1)**
  - Generate performance report in both text and for use with GUI

Report Table Notes Section

- Check the Notes before each table in the text report

```
Notes for table 5:

  The Total value for Process HiMem (MBytes), Process Time is the avg
  for the PE values.

  The value shown for Process HiMem is calculated from information in
  the /proc/self/numa_maps files captured near the end of the program.
  It is the total size of all pages, including huge pages, that were
  actually mapped into physical memory from both private and shared
  memory segments.

  This table shows only the maximum, median, minimum PE entries,
    sorted by Process Time.
```

Report Generation at Scale

CP2K on 200 Intel Broadwell Nodes (7200 MPI Ranks)

Release	Data Processing Time	Report Generation Time
6.4.6	508s	11132s
6.5.0	32s	
7.0.1	15s	261s

- ***pat_report execution time reduced significantly!***
- **Results in less impact on overall job execution**
  - pat_report run at end of job with perftools-lite

Additional Controls for Report Generation

perftools-lite:

- **Optionally run pat_report on the data directory from login node**
  - `export PAT_RT_REPORT_CMD=pat_report,-Q0`
  - `Reduces job execution time, but disables parallel pat_report execution`

perftools-lite or perftools:

- **For a quick preview of performance data, use subset of data to generate a report**
  - `user@login> pat_report -Q1` ← report from 1st ap2 file
  - `user@login> pat_report –Q3` ← report from 1st, middle, and last file

Controlling Instrumentation and Data

- **Record Subset of PEs during execution**
  - It works again! (we found that it was broken last year)
  - Example: export PAT_RT_EXPFILE_PES=0,4,5,10

- **Don't instrument select binaries when using perftools-lite**
  - Good for applications that generate test or intermediate binaries with CMake and GNU Autotools

  - Use **CRAYPAT_LITE_WHITELIST** for binaries you DO want instrumented

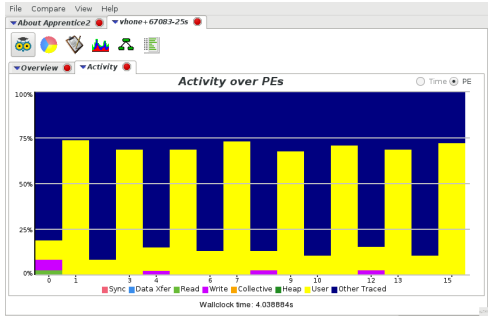
pat_run

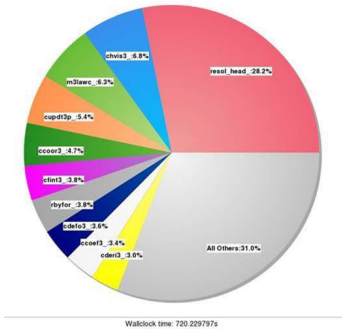
Utility that allows you to profile un-instrumented, dynamically linked binaries with CrayPat!

- **Delivers Cray performance tools profiling information for codes that cannot easily be rebuilt**
- **Makes profiling possible for a wider set of HPC applications**
- **Available starting with perftools 7.0.1**
- **Initially targets Cray XC systems running CLE 6 or later**

Using pat_run

- **Insert before executable in run command**
  - `user@login> srun –n 16 pat_run ./my_program`
  - `user@login> pat_report expdir > my_report`
- **Use existing perftools capability**
  - Optionally collect a different group of performance counters
    - `user@login> export PAT_RT_PERFCTR=1`
    - `user@login> aprun -n 16 pat_run ./my_program`
  - Perform other experiments, for example trace MPI routines
    - `user@login> pat_run -g mpi ./my_program`
  - Create additional views of the data with pat_report options, such as
    - `user@login> pat_report –P –O callers+src`





What About Memory Bandwidth?

- Phased in over perftools 7.0.0, 7.0.1, and 7.0.2 for Intel Xeon processors
- New default counter group with perftools-lite and perftools experiments
- New table for memory bandwidth by NUMA node in default lite and full reports
- Separate functionality from perftools-lite-hbm experiment which uses CCE, CrayPat, and Reveal to tracks memory traffic and associate with allocation sites

Example: Memory Bandwidth per NUMA

8 MPI ranks, 4 on each of 2 nodes

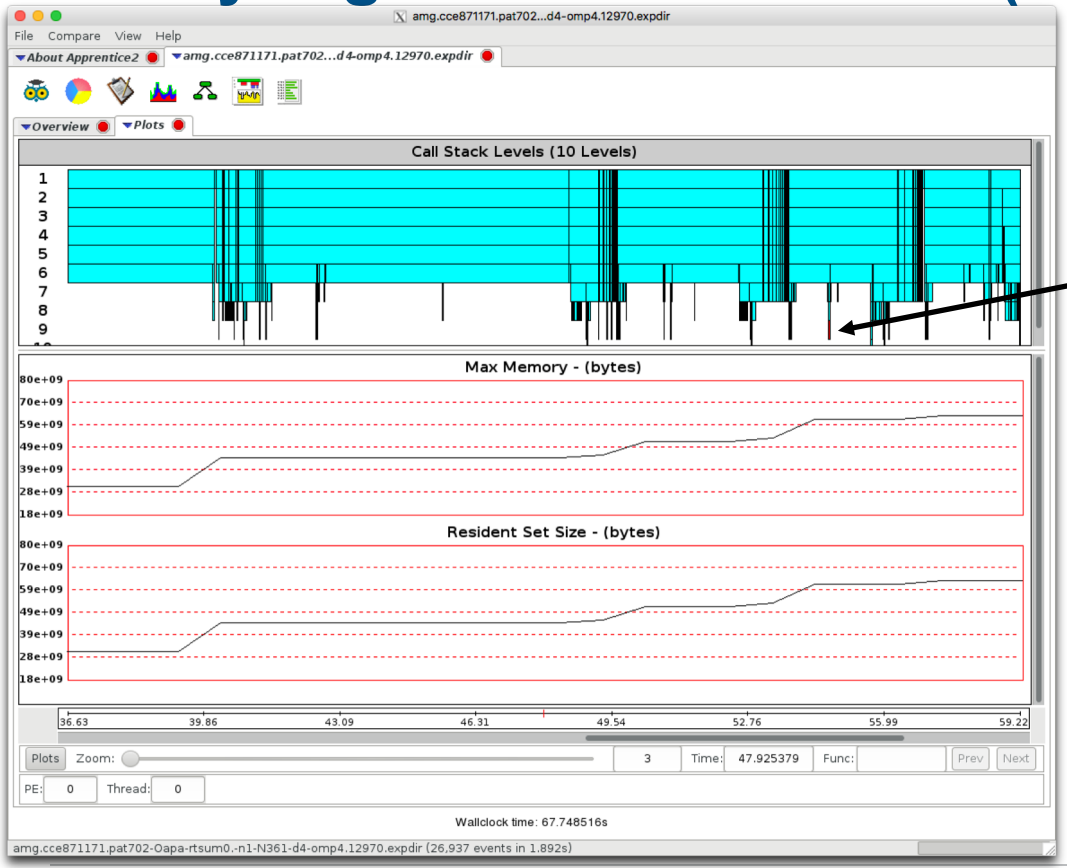
Table 1: Memory Bandwidth by Numanode

Memory Traffic GBytes	Local Memory Traffic GBytes	Remote Memory Traffic GBytes	Thread Time	Memory Traffic GBytes / Sec	Memory Traffic / Nominal Peak	Numanode Node Id PE=HIDE Thread=HIDE
39,429	39,429	0	990.218871	39.82	33.4%	Max of Numanode values
39,429	39,429	0	990.217439	39.82	33.4%	numanode.0
39,429	39,429	0	990.217439	39.82	33.4%	nid.200
38,389	38,389	0	990.224163	38.77	32.5%	nid.205
38,857	38,857	0	990.218903	39.24	32.9%	numanode.1
38,857	38,857	0	990.211194	39.24	32.9%	nid.200
38,528	38,528	0	990.226690	38.91	32.6%	nid.205

Memory Bandwidth Table Tidbits

- **Treat socket as NUMA node**
  - numanode.0 ← number represents socket
- **Max memory speed used in %peak calculation**
  - 7.0.1: hard coded based on theoretical
  - 7.0.2: obtain speed using rca
- **Time is reported per thread**
  - Thread time is lifetime of thread (pure MPI programs report thread0)
- **Traffic split into local and remote with respect to numanode**

Memory High Water Over Time (Apprentice2)



Name:	MPI_Waitall
Start Time:	54.666761
End Time:	54.716777
Group:	MPI

Produced with:

```
pat_build ./my_program
PAT_RT_SAMPLING_DATA=memory
PAT_RT_SUMMARY=0
```

Summary

- Cray performance tools offer functionality that **reduces the time investment** associated with porting and tuning applications on new and existing Cray systems

- Cray performance tools come with a **simple interface plus a wealth of capability** when you need it for analyzing those most critical production codes