Performance Profiling on KNL with Cray perftools-lite

Peter Mendygral
pjm@cray.com
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 publically 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.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and 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 names and brands may be claimed as the property of others. Other product and service names mentioned herein are the trademarks of their respective owners.

Copyright 2018 Cray Inc.
Agenda

- Overview of Cray Performance Tools
- Identifying slowest areas of a program
- Tips for analyzing program performance
Overview of Cray Performance Tools

- Load modules to access software
- Choose experiment to target your goal
- Visualize application bottlenecks
Goals

- 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 your profiling experience by using simple (lite mode) and/or advanced interfaces for a wealth of capability that targets analyzing large HPC jobs
Functional Highlights

- **Whole program performance analysis with**
  - Novice and advanced user interfaces
  - Support for MPI, SHMEM, OpenMP, PGAS, OpenACC, CUDA
  - Load imbalance detection
  - HW counter metrics (hit rates, computational intensity, etc.)
  - Observations on inefficiencies
  - Data correlation to user source
  - Minimal program perturbation

- **Sampling, tracing with runtime summarization (RTS), full trace (timeline) modes available**

- **Supports** CCE, Intel and GCC compilers on Cray XC systems
- **Supports** CCE on Cray CS systems
Components

- **CrayPat and CrayPat-lite**
  - Identifies top time consuming routines, work load imbalance, MPI rank placement strategies, etc.

- **PAPI**
  - Performance counters (used by CrayPat or directly by user)

- **Cray Apprentice2**
  - Visualize load imbalance, excessive communication, network contention, excessive serialization

- **Reveal**
  - View CCE optimization messages, key loops in program, high bandwidth memory traffic, add OpenMP to program
Cray Performance Tools Status

● perftools-base/7.0.0 released in Dec 2017

● What’s new?
  ● Perftools-lite performance improvements (execution, scaling)
  ● Performance data experiment directory
  ● Memory and vector sensitivity metrics
How to Access Cray Performance Tools

**perftools-base module**

- Provides access to performance tools instrumentation modules, documentation, Reveal and Apprentice2

- Doesn’t impact program build

- If not loaded by default on a system, you can load in your .profile or .login and leave it loaded

- Once loaded, do a `module avail perftools` to see available instrumentation modules
Program Instrumentation Modules

- Instrumentation modules prepare an application for performance data collection

> module avail perftools

-------- /opt/cray/pe/perftools/7.0.0/modulefiles --------
perftools
perftools-lite
perftools-lite-events
perftools-lite-gpu
perftools-lite-hbm
perftools-lite-loops
perftools-nwpc

Use first to get basic program performance profile
Interfaces Available

- **CrayPat-lite**: simple interface for convenience

- **CrayPat**: advanced interface 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
Simple vs Advanced Interface

- Has fewer steps

- Is easier to use if you rarely profile applications (don’t have to remember how to use the tools)

- Provides a condensed text report

- Performs performance data processing and report generation at end of job on compute nodes

- Allows you to mix with advanced interface
  - Run `pat_report` to get full report with simple interface
Identifying Slowest Areas of a Program

- Load perftools-lite
- Build and run program
- Interpret results
Example: Using perftools-lite

$ module load perftools-lite

Build program
- Should see message at end of build from CrayPat saying that it created an instrumented executable
- Add \(-\text{hlist}=a\) to build with CCE listing for optimization feedback

$ aprun/srun -n ... ./my_program

Performance data sent to \texttt{STDOUT} and to directory with unique name
- Refer to CCE listing with sampling by line data in Table 2
Example: Cray loopmark Messages

c/c/ftn/cc -hlist=a ...

29. b--------< do i3=2,n3-1
30. b b--------< do i2=2,n2-1
31. b b Vr--< do i1=1,n1
32. b b Vr u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
33. b b Vr * + u(i1,i2,i3-1) + u(i1,i2,i3+1)
34. b b Vr u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
35. b b Vr * + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
36. b b Vr--> enddo
37. b b Vr--< do i1=2,n1-1
38. b b Vr r(i1,i2,i3) = v(i1,i2,i3)
39. b b Vr * - a(0) * u(i1,i2,i3)
40. b b Vr * - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
41. b b Vr * - a(3) * ( u2(i1-1) + u2(i1+1) )
42. b b Vr--> enddo
43. b b------> enddo
44. b-------> enddo

Outer loops were blocked (b)

Inner-loops were vectorized and unrolled (Vr)
Example: Cray loopmark Messages (cont)

ftn-6289 ftn: VECTOR File = resid.f, Line = 29
A loop starting at line 29 was not vectorized because a recurrence was found on "U1" between lines 32 and 38.

ftn-6049 ftn: SCALAR File = resid.f, Line = 29
A loop starting at line 29 was blocked with block size 4.

ftn-6289 ftn: VECTOR File = resid.f, Line = 30
A loop starting at line 30 was not vectorized because a recurrence was found on "U1" between lines 32 and 38.

ftn-6049 ftn: SCALAR File = resid.f, Line = 30
A loop starting at line 30 was blocked with block size 4.

ftn-6005 ftn: SCALAR File = resid.f, Line = 31
A loop starting at line 31 was unrolled 4 times.

ftn-6204 ftn: VECTOR File = resid.f, Line = 31
A loop starting at line 31 was vectorized.

ftn-6005 ftn: SCALAR File = resid.f, Line = 37
A loop starting at line 37 was unrolled 4 times.

ftn-6204 ftn: VECTOR File = resid.f, Line = 37
A loop starting at line 37 was vectorized.
Example of Explain Utility

users/ldr> explain ftn-6289

VECTOR: A loop starting at line %s was not vectorized because a recurrence was found on "var" between lines num and num.

Scalar code was generated for the loop because it contains a linear recurrence. The following loop would cause this message to be issued:

```
DO I = 2,100
    B(I) = A(I-1)
    A(I) = B(I)
ENDDO
```
Example: perftools-lite Job Summary

```bash
CrayPat/X: Version 7.0.0.45 Revision 11f412d 11/08/17 09:36:36
Experiment: lite lite/sample_profile
Number of PEs (MPI ranks): 96
Numbers of PEs per Node: 16 PEs on each of 6 Nodes
Numbers of Threads per PE: 1
Number of Cores per Socket: 68
Execution start time: Tue Nov 14 11:44:06 2017
System name and speed: nid00037 1401 MHz (approx)
Intel Knights Landing CPU Family: 6 Model: 87 Stepping: 1
MCDRAM: 7.2 GHz, 16 GiB available as quad, cache (100% cache)
Avg Process Time: 612.10 secs
High Memory: 16,053.7 MBytes 167.2 MBytes per PE
I/O Read Rate: 1.764988 MBytes/sec
I/O Write Rate: 4.349897 MBytes/sec
```
Example: perftools-lite Top Time Consumers

Table 1: Profile by Function Group and Function (top 10 functions shown)

<table>
<thead>
<tr>
<th>Samp%</th>
<th>Samp</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.0</td>
<td>55,605.7</td>
<td>--</td>
<td>--</td>
<td>Total</td>
</tr>
<tr>
<td>56.5</td>
<td>31,412.8</td>
<td>--</td>
<td>--</td>
<td>USER</td>
</tr>
<tr>
<td>19.7</td>
<td>10,944.1</td>
<td>290.9</td>
<td>2.6</td>
<td>create_boundary$boundary_</td>
</tr>
<tr>
<td>10.7</td>
<td>5,937.8</td>
<td>214.2</td>
<td>3.5</td>
<td>get_block$blocks_</td>
</tr>
<tr>
<td>3.9</td>
<td>2,194.4</td>
<td>7.6</td>
<td>0.3</td>
<td>create_distrb_balanced$distribution_</td>
</tr>
<tr>
<td>2.0</td>
<td>1,135.5</td>
<td>137.5</td>
<td>10.8</td>
<td>impvmixt$vertical_mix_</td>
</tr>
<tr>
<td>1.9</td>
<td>1,064.8</td>
<td>124.2</td>
<td>10.5</td>
<td>impvmixt_correct$vertical_mix_</td>
</tr>
<tr>
<td>22.5</td>
<td>12,513.4</td>
<td>--</td>
<td>--</td>
<td>ETC</td>
</tr>
<tr>
<td>20.1</td>
<td>11,151.4</td>
<td>2,758.6</td>
<td>19.9</td>
<td>__cray_memcpy_KNL</td>
</tr>
<tr>
<td>20.7</td>
<td>11,503.5</td>
<td>--</td>
<td>--</td>
<td>MPI</td>
</tr>
<tr>
<td>11.1</td>
<td>6,171.6</td>
<td>1,785.4</td>
<td>22.5</td>
<td>MPI_ALLREDUCE</td>
</tr>
<tr>
<td>7.9</td>
<td>4,377.8</td>
<td>3,254.2</td>
<td>42.7</td>
<td>mpi_waitall</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Rank Order</th>
<th>On-Node Bytes/PE</th>
<th>On-Node Bytes/PE% of Total</th>
<th>MPICH_RANK_REORDER_METHOD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hilbert</td>
<td>1.413e+12</td>
<td>81.94%</td>
<td>3</td>
</tr>
<tr>
<td>SMP</td>
<td>1.053e+12</td>
<td>61.04%</td>
<td>1</td>
</tr>
<tr>
<td>Fold</td>
<td>9.405e+11</td>
<td>54.53%</td>
<td>2</td>
</tr>
<tr>
<td>RoundRobin</td>
<td>8.962e+11</td>
<td>51.96%</td>
<td>0</td>
</tr>
</tbody>
</table>
### Example: perftools-lite Hot Spots by Line

#### Table 3: Profile by Group, Function, and Line

<table>
<thead>
<tr>
<th>Samp%</th>
<th>Samp</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Group</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100.0%</td>
<td>60,665.8</td>
<td>--</td>
<td>--</td>
<td>Total</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>94.6%</th>
<th>57,390.6</th>
<th>--</th>
<th>--</th>
<th>USER</th>
</tr>
</thead>
<tbody>
<tr>
<td>82.1%</td>
<td>49,835.3</td>
<td>--</td>
<td>--</td>
<td>LAMMPS_NS::PairLJCut::compute</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>----</td>
<td>----</td>
<td>------</td>
</tr>
<tr>
<td>80.7%</td>
<td>48,970.1</td>
<td>--</td>
<td>--</td>
<td>src/Obj_xc30intel/../pair_lj_cut.cpp</td>
</tr>
<tr>
<td>-------</td>
<td>----------</td>
<td>----</td>
<td>----</td>
<td>------</td>
</tr>
<tr>
<td>3.9%</td>
<td>2,359.8</td>
<td>100.2</td>
<td>4.1%</td>
<td>line.102</td>
</tr>
<tr>
<td>1.0%</td>
<td>596.2</td>
<td>61.8</td>
<td>9.5%</td>
<td>line.105</td>
</tr>
<tr>
<td>8.3%</td>
<td>5,022.4</td>
<td>683.6</td>
<td>12.1%</td>
<td>line.107</td>
</tr>
<tr>
<td>2.9%</td>
<td>1,744.2</td>
<td>966.8</td>
<td>36.0%</td>
<td>line.108</td>
</tr>
</tbody>
</table>
Get Additional Information Without Re-running

- Run `pat_report` after collecting data with lite mode
  - `pat_report my_programXXXs/ > full_rpt`
  - `pat_report -O callers` or `pat_report -O callers+src`
  - `pat_report -O calltree` or `pat_report -O calltree+src`
  - Check out load balance table

- Learn about related tables in "Table Notes"
  - We try to suggest reports that dive deeper on a related topic
  - Provide data aggregation method
Example: Load Balance by Max Time

Table 2: Profile of maximum function times (limited entries shown)

<table>
<thead>
<tr>
<th>Samp%</th>
<th>Samp</th>
<th>Imb.</th>
<th>Imb.</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PE=[max,min]</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>100.0%</td>
<td>51,891.0</td>
<td>2,055.7</td>
<td>4.0%</td>
<td>LAMMPS_NS::PairLJCut::compute</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>100.0%</td>
<td>51,891.0</td>
<td>--</td>
<td>--</td>
<td>pe.32</td>
</tr>
<tr>
<td>93.0%</td>
<td>48,263.0</td>
<td>--</td>
<td>--</td>
<td>pe.93</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>11.3%</td>
<td>5,871.0</td>
<td>193.1</td>
<td>3.3%</td>
<td>LAMMPS_NS::Neighbor::half_bin_newton</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>11.3%</td>
<td>5,871.0</td>
<td>--</td>
<td>--</td>
<td>pe.66</td>
</tr>
<tr>
<td>10.7%</td>
<td>5,535.0</td>
<td>--</td>
<td>--</td>
<td>pe.94</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>8.6%</td>
<td>4,480.0</td>
<td>2,418.6</td>
<td>54.6%</td>
<td>MPI_Send</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>8.6%</td>
<td>4,480.0</td>
<td>--</td>
<td>--</td>
<td>pe.45</td>
</tr>
<tr>
<td>0.9%</td>
<td>443.0</td>
<td>--</td>
<td>--</td>
<td>pe.32</td>
</tr>
</tbody>
</table>
Recognizing OpenMP in a Report (CCE only)

- CrayPat can collect the most detail from OpenMP using the Cray compiler

- OpenMP regions and loops are identified in report with the following syntax:
  - function.REGION@li.49
  - function.LOOP@li.53

- OpenMP statistics are collected by default (no need to enable anything in the tools)
  - Most information is available with Cray compiler
How Do I See per-Rank or per-Thread Data?

- \$ pat_report \(-s\) pe=ALL

- \$ pat_report \(-s\) th=ALL
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
Documentation Available

- Release Notes
  - > module help perftools-base/version_number


- pat_help – interactive help utility on the Cray Performance toolset

- Man pages
Tips for Analyzing Program Performance

Load perftools-lite

Build and run program

Interpret results
Where Do I Start?

- Determine problem size / job size that you ultimately want to run

- Get high level program profile at scale to locate key bottlenecks

- Work from high-level (inter-node) to low-level (intra-node) bottlenecks
Helpful 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 loop optimization, including adding OpenMP with Reveal (CCE only)**
  - Use `perftools-lite-loops`
  - Use `perftools-lite-hbm` for memory bandwidth sensitivity study

- **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
  - Good for scaling analysis at targeted final job size
Focus on Loop Optimization – Find Top Loops

- $ module load PrgEnv-crty perftools-lite-loops
  - Needs Cray compiler

- **Build program (build from scratch – we add compiler flags)**
  - Should see message at end of build from CrayPat 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

- $ aprun -n ... ./my_program

- Performance data sent to STDOUT and to experiment data directory with unique name
Focus on MPI Communication

- \$ module load perftools

- **Build program**
  - Remember to add \( -\text{hlist}=a \) to build with CCE listing
  - Can relink or use “a.out+orig” if created with perftools-lite

- **Instrument program**
  - \$ pat_build \(-g\) mpi ./my_program

- **Run application**
  - \$ aprun/srun \(-n\) ... my_program+pat

- **Create report**
  - \$ pat_report my_programXXX/ > my_report
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.
Questions?

Thank You!