BG/Q Performance Tools

Scott Parker
Leap to Petascale Workshop: May 22-25, 2012
Argonne Leadership Computing Facility
BG/Q Performance Tool Development

- In conjunction with the Early Science program an Early Software efforts was initiated to bring widely used performance tools, debuggers, and libraries to the BG/Q

- Motivation was to take lesson learned from P and applied to Q:
  - Tools and libraries were not in the shape we wanted in the early days of P
  - Don’t rely solely on vendor tools
  - A number of useful and popular open source HPC tools exist
  - Development best done as a collaboration between vendor, developers, and users
  - Work with the vendor (IBM) as early as possible in the process:
    - vendor doesn’t always have a clear idea of tools/user requirements
    - vendor expertise will dissipate after development phase ends
  - Engage the tools community early for requirements and porting
  - Insure functionality of vendor base software layer supporting tools:
    - hardware performance counter API, stack unwinding, handling interrupts and traps, debugging and symbol table information, debugger process control, compiler instrumentation
## BG/Q Tools Development Status

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Source</th>
<th>Provides</th>
<th>Q Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>bgpm</td>
<td>IBM</td>
<td>HPC</td>
<td>Available</td>
</tr>
<tr>
<td>gprof</td>
<td>GNU/IBM</td>
<td>Timing (sample)</td>
<td>Available</td>
</tr>
<tr>
<td>TAU</td>
<td>Univ. Oregon</td>
<td>Timing (inst, sample), MPI, HPC</td>
<td>Available</td>
</tr>
<tr>
<td>Rice HPCToolkit</td>
<td>Rice Univ.</td>
<td>Timing (sample), HPC (sample)</td>
<td>Available</td>
</tr>
<tr>
<td>IBM HPCT</td>
<td>IBM</td>
<td>MPI, HPC</td>
<td>In development. Beta available</td>
</tr>
<tr>
<td>mpiP</td>
<td>LLNL</td>
<td>MPI</td>
<td>Available</td>
</tr>
<tr>
<td>PAPI</td>
<td>UTK</td>
<td>HPC API</td>
<td>Available</td>
</tr>
<tr>
<td>Darshan</td>
<td>ANL</td>
<td>IO</td>
<td>Available</td>
</tr>
<tr>
<td>Open</td>
<td>Speedshop</td>
<td>Krell</td>
<td>Timing (sample), HCP, MPI, IO</td>
</tr>
<tr>
<td>Scalasca</td>
<td>Juelich</td>
<td>Timing (inst), MPI</td>
<td>Available</td>
</tr>
<tr>
<td>DynInst</td>
<td>UMD/Wisc/IBM</td>
<td>Binary rewriter</td>
<td>In development</td>
</tr>
<tr>
<td>ValGrind</td>
<td>ValGrind/IBM</td>
<td>Memory &amp; Thread Error Check</td>
<td>Development pending</td>
</tr>
<tr>
<td>Jumpshot</td>
<td>ANL</td>
<td>MPI</td>
<td>Available</td>
</tr>
</tbody>
</table>
Aspects of providing tools on BG/Q

- **The Good** - BG/Q provides a hardware & software environment that enables many standard performance tools to be provided:
  - Software:
    - Environment similar to 64 bit PowerPC Linux: Provides standard GNU binutils
    - New, much-improved performance counter API bgpm
  - Improved Performance Counter Hardware:
    - BG/Q provides 424 64-bit counters in a central node counting unit for all hardware units: processor cores, prefetchers, L2 cache, memory, network, message unit, PCIe, DevBus, and CNK events
    - Countable events include: instruction counts, flop counts, cache events, and many more
    - Provides support for hardware threads and counters can be controlled at the thread level

- **The Bad** – some aspects of the hardware/software environment are unique to BG/Q
  - O/S ‘Linux like’ but not really Linux:
    - doesn’t support all Linux system calls/features: doesn’t have fork(), /proc, shell
  - static linking by default
  - non-standard instruction set (addition of quad floating point SIMD inst.)
  - hardware performance counter setup is unique to BG/Q
  - to get tools working early in BG/Q lifecycle required working in pre-production environment
# Tools and API’s Available on Cetus

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Source</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGPM</td>
<td>IBM</td>
<td>HW Counter API</td>
</tr>
<tr>
<td>PAPI</td>
<td>UTK</td>
<td>HW Counter API</td>
</tr>
<tr>
<td>gprof</td>
<td>GNU/IBM</td>
<td>Timing (sample)</td>
</tr>
<tr>
<td>TAU</td>
<td>Univ. Oregon</td>
<td>Timing (inst, sample), MPI, HW Counters (inst)</td>
</tr>
<tr>
<td>Rice HPCToolkit</td>
<td>Rice Univ.</td>
<td>Timing (sample), HW Counters (sample)</td>
</tr>
<tr>
<td>Scalasca</td>
<td>Juelich</td>
<td>Timing (inst), MPI</td>
</tr>
<tr>
<td>OpenSpeedShop</td>
<td>Krell</td>
<td>Timing (sample), HCP, MPI, IO</td>
</tr>
<tr>
<td>IBM HPCT</td>
<td>IBM</td>
<td>MPI, HW Counters</td>
</tr>
<tr>
<td>mpiP</td>
<td>LLNL</td>
<td>MPI</td>
</tr>
<tr>
<td>Darshan</td>
<td>ANL</td>
<td>IO</td>
</tr>
<tr>
<td>Jumpshot</td>
<td>ANL</td>
<td>MPI</td>
</tr>
</tbody>
</table>
What to expect from performance tools

- Performance tools collect information during the execution of a program to enable the performance of an application to be understood, documented, and improved.
- A wide variety of performance tools exist which collect different information in different ways.
- It is up to the user to determine:
  - what tool to use
  - what information to collect
  - how to interpret the collected information
  - how to change code to improve the performance
Types of Performance Information

- Types of information collected by performance tools:
  - Time in routines and code sections
  - Call counts for routines
  - Call graph information with timing attribution
  - Information about arguments passed to routines:
    - MPI – message size
    - IO – bytes written or read
  - Hardware performance counter data:
    - FLOPS
    - Instruction counts (total, integer, floating point, load/store, branch, ...)
    - Cache hits/misses
    - Memory, bytes loaded and stored
    - Pipeline stall cycle
  - Memory usage
Sources of Performance Information

- **Code Instrumentation:**
  - Insert calls to performance collection routines into the code to be analyzed
  - Allows a wide variety of data to be collected
  - Source instrumentation can only be used on routines that can be edited and compiled, may not be able to instrument libraries
  - Can be done: manually, by the compiler, or automatically by a parser, or after compilation by a binary instrumenter
Sources of Performance Information

- Execution Sampling:
  - Requires virtually no changes made to program
  - Execution of the program is halted periodically by an interrupt source (usually a timer) and location of the program counter is recorded and optionally call stack is unwound
  - Allows attribution of time (or other interrupt source) to routines and lines of code based on the number of time the program counter at interrupt observed in routine or at line
  - Estimation of time attribution is not exact but with enough samples error is negligible
  - Require debugging information in executable to attribute below routine level
  - Performance data can be collected for entire program including libraries
Sources of Performance Information

- Library interposition:
  - A call made to a function is intercepted by a wrapping performance tool routine
  - Allows information about the intercepted call to captured and recorded, including: timing, call counts, and argument information
  - Can be done for any routine by one of several methods:
    - Some libraries (like MPI) are designed to allow calls to be intercepted. This is done using weak symbols (MPI_Send) and alternate routine names (PMPI_Send)
    - LD_PRELOAD – can force shared libraries to be loaded from alternate locations containing interposing routines, which then call actual routine
    - Linker Wrapping – instruct the linker to resolve all calls to a routine (ex: malloc) to an alternate wrapping routine (ex: malloc_wrap) which can then call the original routine
Sources of Performance Information

- **Hardware Performance Counters:**
  - Hardware register implemented on the chip that record counts of performance related events. For example:
    - FLOPS - Floating point operations
    - L1 cache miss – number of L1 requests that miss
  - Processors support a limited set of counters and countable events are preset by chip designer
  - Counters capabilities and events differ significantly between processors
  - Need to use an API to configure, start, stop, and read counters
  - Blue Gene/Q:
    - Has 424 performance counter registers for: core, L1, prefetcher, L2 cache, memory, network, message unit, PCIe, DevBus, and CNK events
## Tools and API’s Available on Cetus

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>Source</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGPM</td>
<td>IBM</td>
<td>HW Counter API</td>
</tr>
<tr>
<td>PAPI</td>
<td>UTK</td>
<td>HW Counter API</td>
</tr>
<tr>
<td>gprof</td>
<td>GNU/IBM</td>
<td>Timing (sample)</td>
</tr>
<tr>
<td>TAU</td>
<td>Unv. Oregon</td>
<td>Timing (inst, sample), MPI, HW Counters (inst)</td>
</tr>
<tr>
<td>Rice HPCToolkit</td>
<td>Rice Univ.</td>
<td>Timing (sample), HW Counters (sample)</td>
</tr>
<tr>
<td>Scalasca</td>
<td>Juelich</td>
<td>Timing (inst), MPI</td>
</tr>
<tr>
<td>OpenSpeedShop</td>
<td>Krell</td>
<td>Timing (sample), HCP, MPI, IO</td>
</tr>
<tr>
<td>IBM HPCT</td>
<td>IBM</td>
<td>MPI, HW Counters</td>
</tr>
<tr>
<td>mpiP</td>
<td>LLNL</td>
<td>MPI</td>
</tr>
<tr>
<td>Darshan</td>
<td>ANL</td>
<td>IO</td>
</tr>
<tr>
<td>Jumpshot</td>
<td>ANL</td>
<td>MPI</td>
</tr>
</tbody>
</table>
BGPM API

- Hardware performance counters collect counts of operations and events at the hardware level
- BGPM is the C programming interface to the hardware performance counters
  - Manages complexity of underlying hardware through simple API
  - Provides multiple modes including: hardware distributed, software distributed, and low latency
  - Allows thread level control of core level counters
- Counter Hardware:
  - Centralized set of registers collecting counts from distributed units
    - Sixteen A2 CPU cores, L1P and Wakeup units
    - Sixteen L2 units (memory slices)
    - Message Unit
    - PCIe Unit
    - Devbus
  - Separate counter units for: Memory Controller, Network
BGPM Hardware Counters
BGPM Hardware Counters

A2 Processor Core

FU Event Selection
T0_Events(0:7)
T1_Events(0:7)
T2_Events(0:7)
T3_Events(0:7)

XU Event Selection
T0_Events(0:31)
T1_Events(0:31)
T2_Events(0:31)
T3_Events(0:31)

LSU Event Selection
T0_Events(0:31)
T1_Events(0:31)
T2_Events(0:31)
T3_Events(0:31)

MMU Event Selection
T0_Events(0:15)
T1_Events(0:15)
T2_Events(0:15)
T3_Events(0:15)

Config Regs

Switch
6 groups

24 14-bit local counters
6 per thread

dbg bus

Op code decode

UPC_P Unit

L1P/Wakeup Units

Basic(0:48)
Wakeup(0:7)
Switch(0:34)
Stream(0:34)
List(0:34)

Select

~162 events

480 total events

Argonne Leadership Computing Facility
BGPM API

- **Example BGPM calls:**

  ```c
  Bgpm_Init(BGPM_MODE_SWDISTRIB);
  int hEvtSet = Bgpm_CreateEventSet();
  unsigned evtList[] = { PEVT_IU_IS1_STALL_CYC, PEVT_IU_IS2_STALL_CYC,
                        PEVT_CYCLES, PEVT_INST_ALL, };;
  Bgpm_AddEventList(hEvtSet, evtList, sizeof(evtList)/sizeof(unsigned) );
  Bgpm_Apply(hEvtSet);
  Bgpm_Start(hEvtSet);
  Workload();
  Bgpm_Stop(hEvtSet)
  Bgpm_ReadEvent(hEvtSet, i, &cnt);
  printf(" 0x%016lx <= %s\n", cnt, Bgpm_GetEventLabel(hEvtSet, i));
  ```

- **Installed in:** /bgsys/drivers/ppcfloor/bgpm/

- **Documentation:**
  - /bgsys/drivers/ppcfloor/bgpm/docs
## BGPM Events

### Punit Events

<table>
<thead>
<tr>
<th>EventId</th>
<th>Label</th>
<th>Description</th>
<th>Scope</th>
<th>Features</th>
<th>Tag</th>
<th>PAPI</th>
<th>Status</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PEVT_UNDEF</td>
<td>undefined</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>Undefined event should not occ...</td>
</tr>
<tr>
<td></td>
<td><strong>AXU Execution Unit Events (Quad Floating Point Unit)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>PEVT_AXU_INSTR_COMMIT</td>
<td>AXU Instruction Committed</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>A valid AXU (non-load/store) I...</td>
</tr>
<tr>
<td>2</td>
<td>PEVT_AXU_CR_COMMIT</td>
<td>AXU CR Instruction Committed</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>A valid AXU CR updater instruc...</td>
</tr>
<tr>
<td>3</td>
<td>PEVT_AXU_IDLE</td>
<td>AXU Idle</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>No valid AXU instruction is in...</td>
</tr>
<tr>
<td>4</td>
<td>PEVT_AXU_FP_DS_ACTIVE</td>
<td>AXU FP Divide or Square root in progress</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>A Floating-Point Divide or Squ...</td>
</tr>
<tr>
<td>5</td>
<td>PEVT_AXU_FP_DS_ACTIVE_CYC</td>
<td>AXU FP Divide or Square root in progress cycles</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>Number of Cycles for a Floatin...</td>
</tr>
<tr>
<td>6</td>
<td>PEVT_AXU_DENORM_FLUSH</td>
<td>AXU Denormal Operand flush</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>A B operand of a Floating Point...</td>
</tr>
<tr>
<td>7</td>
<td>PEVT_AXU_UCODE_OPS_COMMIT</td>
<td>AXU uCode Operations Committed</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>A valid AXU ucode operation is...</td>
</tr>
<tr>
<td>8</td>
<td>PEVT_AXU_FP_EXCEPT</td>
<td>AXU Floating Point Exception</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>FP Exception – FX bit of the F...</td>
</tr>
<tr>
<td>9</td>
<td>PEVT_AXU_FP_ENAB_EXCEPT</td>
<td>AXU Floating Point Enabled Exception</td>
<td>thread</td>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td>FP Enabled Exception – FEX bit...</td>
</tr>
</tbody>
</table>
PAPI

- The Performance API (PAPI) specifies a standard API for accessing hardware performance counters available on most modern microprocessors.

- PAPI provides two interfaces to counter hardware:
  - simple high level interface
  - more complex low level interface providing more functionality

- Defines two classes of hardware events (run `papi_avail`):
  - PAPI Preset Events – Standard predefined set of events that are typically found on many CPU’s. Derived from one or more native events. (e.g., PAP_FP_OPS – count of floating point operations)
  - Native Events – Allows access to all platform hardware counters (e.g., PEVT_XU_COMMIT – count of all instructions issued)

- Installed in `/soft/perftools/papi`

**gprof**

- Widely available Unix tool for collecting timing and call graph information via sampling

Collects information on:
- Approximate time spent in each routine
- Count of the number times a routine was invoked
- Call graph information:
  - list of the parent routines that invoke a given routine
  - list of the child routines a given routine invokes
  - estimate of the cumulative time spent in the child routines

Advantages: widely available, easy to use, robust

Disadvantages: scalability, opens one file per rank, doesn’t work with threads
Using gprof:

- Compile all and link all routines with the ‘-pg’ flag
  
  `mpixlc -pg -g -O2 test.c ...`
  
  `mpixlf90 -pg -g -O2 test.f90 ...`

- Run the code as usual: will generate one gmon.out for each rank (careful!)

- View data in gmon.out files, by running:
  
  `gprof <executable-file> gmon.out.<id>`

  - flags:
    
    - `-p`: display flat profile
    
    - `-q`: display call graph information
    
    - `-l`: displays instruction profile at source line level instead of function level
    
    - `-C`: display routine names and the execution counts obtained by invocation profiling
**TAU**

- The TAU (Tuning and Analysis Utilities) Performance System is a portable profiling and tracing toolkit for performance analysis of parallel programs.
- TAU gathers performance information while a program executes through instrumentation of functions, methods, basic blocks, and statements via:
  - Automatic instrumentation of the code at the source level using the Program Database Toolkit (PDT)
  - Automatic instrumentation of the code using the compiler
  - Manual instrumentation using the instrumentation API
  - At runtime using library call interception
  - Runtime sampling
Some of the types of information that TAU can collect:

- time spent in each routine
- the number of times a routine was invoked
- counts from hardware performance counters via PAPI
- MPI profile information
- Pthread, OpenMP information
- memory usage

Installed in /soft/perftools/tau

Jumpshot

- Jumpshot is an interactive tool for visualizing parallel program behavior
- It can be used to acquire understanding of what a program is doing and why
- It can be used as one of multiple ways to process TAU data
- Events can be inspected at multiple scales
Rice HPCToolkit

- A performance toolkit that utilizes statistical sampling for measurement and analysis of program performance
- Assembles performance measurements into a call path profile that associates the costs of each function call with its full calling context
- Samples timers and hardware performance counters
  - Time bases profiling is supported on the Blue Gene/Q
- Traces can be generated from a time history of samples
- Viewer provides multiple views of performance data including: top-down, bottom-up, and flat views
- Installed in /soft/perftools/hpctoolkit
Scalasca

- Designed for scalable performance analysis of large scale parallel applications
- Instruments code and can collect and provides summary and trace information
- Provides automatic even trace analysis for inefficient communication patterns and wait states
- Graphical viewer for reviewing collected data
- Support for MPI, OpenMP, and hybrid MPI-OpenMP codes
- Installed in /soft/perftools/scalasca
Open|SpeedShop

- Toolkit that collects information using sampling and library interposition, including:
  - Sampling based on time or hardware performance counters
  - MPI profiling and tracing
  - Hardware performance counter data
  - IO profiling and tracing
- GUI and command line interfaces for viewing performance data
- Installed in /soft/perftools/oss
IBM HPCT

- A package from IBM that includes several performance tools:
  - MPI Profile and Trace library
  - Hardware Performance Monitor (HPM) API

- Consists of three libraries:
  - libmpitrace.a – provides information on MPI calls and performance
  - libmpihpm.a – provides MPI and hardware counter data
  - libmpihpm_smp.a – provides MPI and hardware counter data for threaded code

- Installed in: /soft/perftools/hpctw/

IBM HPCT - MPI Trace

- MPI Profiling and Tracing library
  - Collects profile and trace information about the use of MPI routines during a program's execution via library interposition
  - Profile information collected:
    - Number of times an MPI routine was called
    - Time spent in each MPI routine
    - Message size histogram
  - Tracing can be enabled and provides a detailed time history of MPI events
  - Point-to-Point communication pattern information can be collected in the form of a communication matrix
  - By default MPI data is collected for the entire run of the application. Can isolate data collection using API calls:
    - `summary_start()`
    - `summary_stop()`
IBM HPCT - MPI Trace

- Using the MPI Profiling and Tracing library
  - Recompile the code to include the profiling and tracing library:
    ```
    mpixlc -o test-c test.c -g -L/soft/perftools/hpctw-lmpitrace
    mpixlf90 -o test-f90 test.f90 -g -L/soft/perftools/hpctw -lmpitrace
    ```
  - Run the code as usual, optionally can control behavior using environment variables.
    - `SAVE_ALL_TASKS={yes,no}`
    - `TRACE_ALL_EVENTS={yes,no}`
    - `PROFILE_BY_CALL_SITE={yes,no}`
    - `TRACEBACK_LEVEL={1,2..}`
    - `TRACE_SEND_PATTERN={yes,no}`
IBM HPCT - MPI Trace

- MPI Trace output:
  - Produces files name mpi_profile.<rank>
  - Default output is MPI information for 4 ranks – rank 0, rank with max MPI time, rank with MIN MPI time, rank with MEAN MPI time. Can get output from all ranks by setting environment variables.

Data for MPI rank 0 of 1024
Times and statistics from MPI_Init() to MPI_Finalize().

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>3</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>5</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>1023</td>
<td>192.0</td>
<td>0.005</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>5115</td>
<td>206.4</td>
<td>0.003</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>5</td>
<td>62125.3</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>3</td>
<td>0.0</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>3</td>
<td>0.0</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>4</td>
<td>3925.0</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>5</td>
<td>15.2</td>
<td>0.000</td>
</tr>
</tbody>
</table>

MPI task 0 of 1024 had the maximum communication time.
total communication time = 1.677 seconds.
total elapsed time = 38.647 seconds.
heap memory used = 38.844 Mbytes.
IBM HPCT - HPM

- Hardware Performance Monitor (HPM) is an IBM library and API for accessing hardware performance counters
  - Configures, controls, and reads hardware performance counters
- Can select the counter used by setting the environment variable:
  - HPM_GROUP= 0,1,2,3 ...
- By default hardware performance data is collected for the entire run of the application
- API provides calls to isolate data collection.

```c
void HPM_Start(char *label) - start counting in a block marked by label
void HPM_Stop(char *label) - stop counting in a block marked by label
```
IBM HPCT - HPM

- To Use:
  - Link with HPM library:

    ```
    
    ```
  - Run the code as usual, setting environments variables as needed
IBM HPCT - HPM

- HPM output:
  - files output at program termination are:
    - hpm_summary.<rank> – hpm files containing counter data, summed over node

```
Hardware counter report for BGQ - sum for node <0,0,0,0,0,0).
cores in use = 16, active threads per core = 4.

--- Counter values summed over processes on this node ----
  0 162799223298 Committed Load Misses
  0 146867542314 Committed Cacheable Loads
  0 15993740380 L1 miss
  0 476269353705 All XU Instruction Completions
  0 224066635616 All AXU Instruction Completions
  0 444560789065 FP Operations Group 1

-- L2 counters (shared for the node) -------------------------------
  100 61653146465 L2 Hits
  100 383554 L2 Misses
  100 190738 L2 Lines loaded from main memory
  100 368363 L2 Lines stored to main memory

Derived metrics for code block "mpiAll" averaged over process(es) on node <0,0,0,0,0,0>:
  Instruction mix:  FPU = 31.99 %, FXU = 68.01 %
  Instructions per cycle completed per core = 0.7079
  Per cent of max issue rate per core = 48.14 %
  Total weighted GFlops for this node = 11.503
  Loads that hit in L1 d-cache = 88.92 %
  L1 cache = 0.22 %
  L2 cache = 10.86 %
  DDR = 0.00 %
  DDR traffic for the node: ld = 0.000, lt = 0.001, total = 0.001 (Bytes/cycle)
```
mpiP

- mpiP is a lightweight profiling library for MPI applications
- mpiP provides MPI information broken down by program call site:
  - the time spent in various MPI routines
  - the number of time an MPI routine was called
  - information on message sizes
- Writes a single summary file containing MPI information on job completion
- Installed in: /soft/perftools/mpiP
- Documentation at: wiki.alcf.anl.gov/bgq-earlyaccess/index.php/MpiP
Using mpiP:

- Recompile the code to include the mpiP library

  ```
  ```

  ```
  ```

- Run the code as usual, optionally can control behavior using environment variable MPIP

- Output is a single .mpiP summary file containing MPI information for all ranks
Darshan

- Darshan is a library that collects information on a program's IO operations and performance
  - Uses library interception for POSIX and MPI IO calls
- Unlike BG/P, Darshan is not enabled by default yet
- To use Darshan, add the appropriate `softenv` keys:
  - `+darshan-xl`
  - `+darshan-gcc`
  - `+darshan-xl.ndebug`
- Upon successful job completion, Darshan data is written to a file named
  - `<USERNAME>_<BINARY_NAME>_<COBALT_JOB_ID>_<DATE>.darshan.gz`
  - in the directory:
    - `/veas-fs0/logs/darshan/<year>/<month>/<day>`
Steps in looking at performance

- Time based profile with at least one of the following:
  - Rice HPCToolkit
  - TAU
  - Scalasca
  - gprof

- MPI profile with:
  - Scalasca
  - Tau
  - HPCTW MPI Profiling Library
  - mpiP
  - Jumpshot

- Gather performance counter information for critical routines:
  - PAPI
  - HPCTW