Argonne NATIONAL LABORATORY

BG/Q Performance Tools

Scott Parker

Argonne Leadership Computing Facility

Extreme Scale Computing Training Program

August 5th, 2013

U.S. DEPARTMENT OF ENERGY

Tools and API's Available on Mira and Vesta

Tool Name	Source	Provides
BGPM	IBM	HW Counter API
PAPI	UTK	HW Counter API
gprof	GNU/IBM	Timing (sample), call graphs (inst)
TAU	Unv. Oregon	Timing (inst, sample), MPI (intercept), HW Counters (inst)
Rice HPCToolkit	Rice Unv.	Timing (sample), HW Counters (sample)
Scalasca	Juelich	Timing (inst), MPI (intercept)
OpenSpeedShop	Krell	Timing (sample), HW Counter (inst), MPI (intercept), IO (intercept)
IBM HPCT	IBM	MPI (intercept), HW Counter info (inst)
mpiP	LLNL	MPI (intercept)
Darshan	ANL	IO (intercept)

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
- Different tools collect different information and collect it 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
  - API used 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

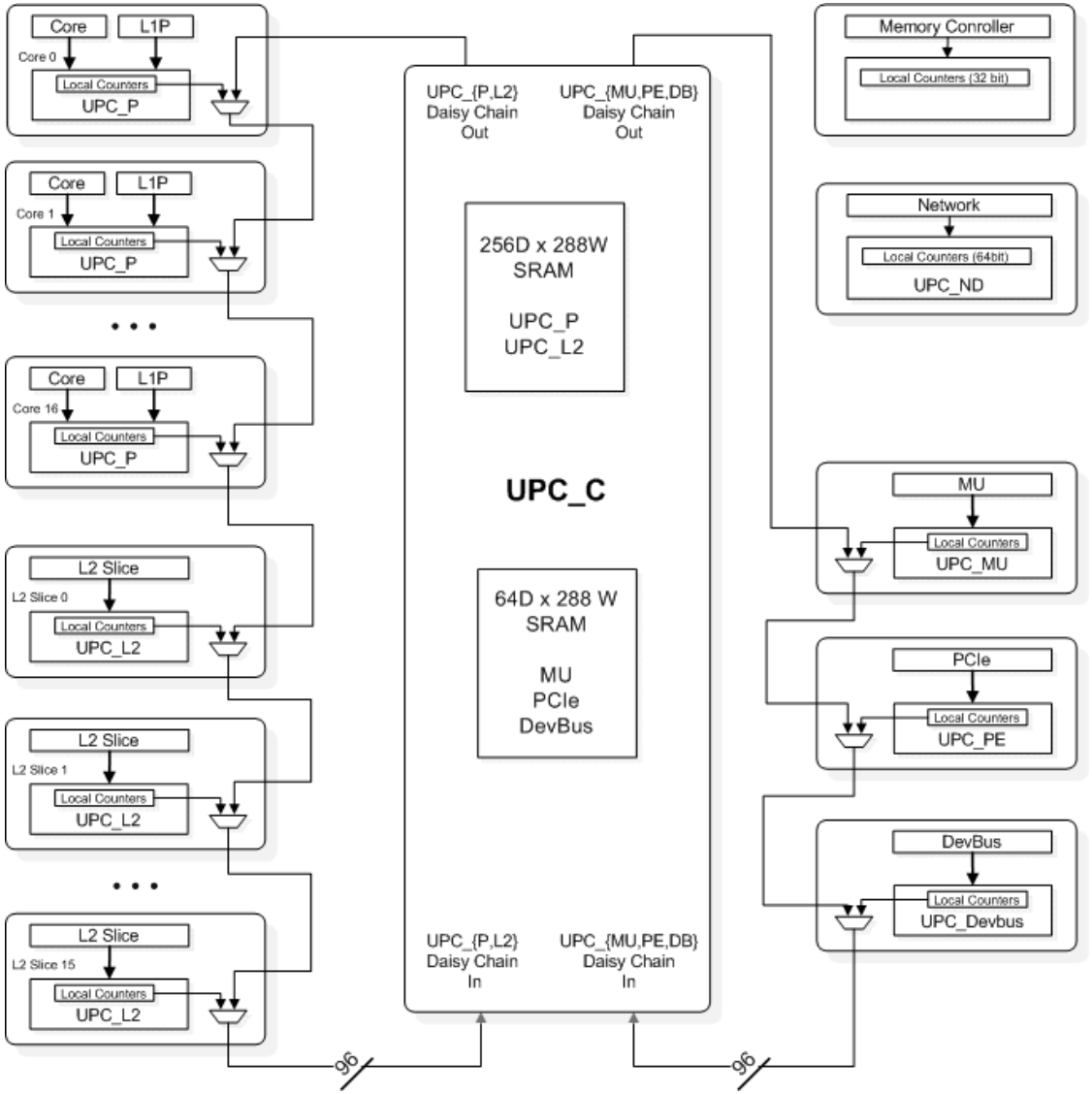
Tools and API's Available on Mira

Tool Name	Source	Provides
BGPM	IBM	HW Counter API
PAPI	UTK	HW Counter API
gprof	GNU/IBM	Timing (sample), call graphs (inst)
TAU	Unv. Oregon	Timing (inst, sample), MPI (intercept), HW Counters (inst)
Rice HPCToolkit	Rice Unv.	Timing (sample), HW Counters (sample)
Scalasca	Juelich	Timing (inst), MPI (intercept)
OpenSpeedShop	Krell	Timing (sample), HW Counter (inst), MPI (intercept), IO (intercept)
IBM HPCT	IBM	MPI (intercept), HW Counter info (inst)
mpiP	LLNL	MPI (intercept)
Darshan	ANL	IO (intercept)

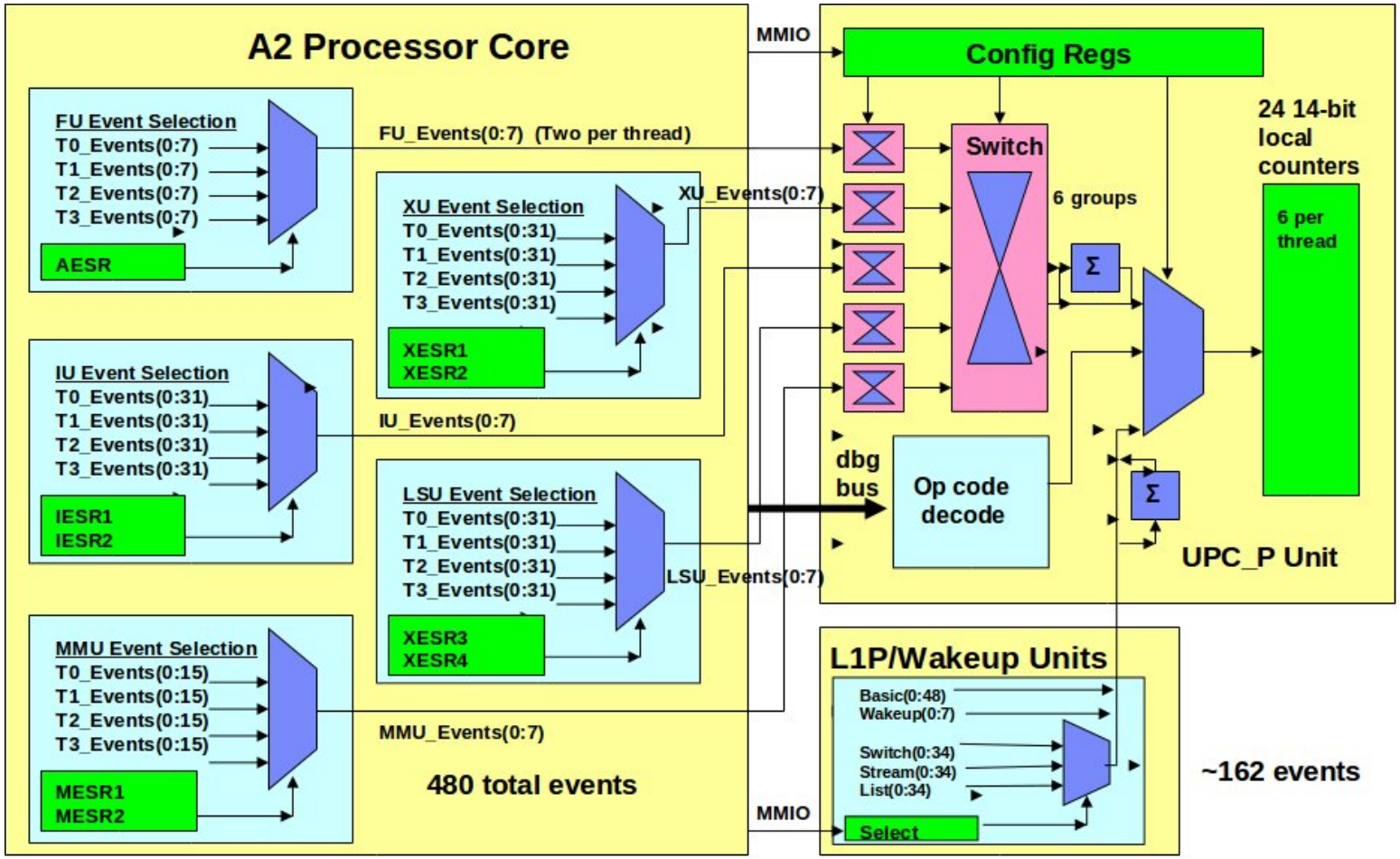
BGPM API

- Hardware performance counters collect counts of operations and events at the hardware level
- BGPM is the C programming interface to the hardware performance counters
- 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



BGPM API

- Example BGPM calls:

```
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/
  - /soft/perftools/bpgm (*convenience link*)
- Documentation:
  - www.alcf.anl.gov/user-guides/bgq-performance-counters
  - */bgsys/drivers/ppcfloor/bgpm/docs*

BGPM Events

Punit Events

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

PAPI

- The Performance API (PAPI) specifies a standard API for accessing hardware performance counters available on most 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. (ex: PAP_FP_OPS – count of floating point operations)
    - Native Events – Allows access to all platform hardware counters (ex: PEVT_XU_COMMIT – count of all instructions issued)
- Installed in /soft/perftools/papi
- Documentation:
  - www.alcf.anl.gov/user-guides/papi
  - *icl.cs.utk.edu/papi/*

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 and sampling 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

TAU

- 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
- Documentation:
  - *www.alcf.anl.gov/user-guides/tuning-and-analysis-utilities-tau*
  - *www.cs.uoregon.edu/Research/tau/docs.php*

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
- Documentation:
  - *www.alcf.anl.gov/user-guides/hpctoolkit*
  - *hpctoolkit.org/documentation.html*

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
  - libhpmprof.a – sampling using hardware counters
- Installed in: /soft/perftools/hpctw/
- Documentation:
  - *www.alcf.anl.gov/user-guides/hpctw*
  - */soft/perftools/hpctw/doc/*

IBM HPCT - MPI Trace

- MPI Profiling and Tracing library
  - Collects profile and trace information about the use of MPI routines during a programs 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/lib -lmpitrace
mpixlf90 -o test-f90 test.f90 -g -L/soft/perftools/hpctw/lib -
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().
---------------------------------------------------------------
MPI Routine                 #calls     avg. bytes      time(sec)
---------------------------------------------------------------
MPI_Comm_size                    3            0.0          0.000
MPI_Comm_rank                    5            0.0          0.000
MPI_Send                      1023          192.0          0.005
MPI_Irecv                     5115          206.4          0.003
MPI_Waitall                      5            0.0          1.668
MPI_Bcast                        3        62125.3          0.001
MPI_Barrier                      3            0.0          0.000
MPI_Reduce                       4         3925.0          0.001
MPI_Allreduce                    5           15.2          0.000
---------------------------------------------------------------
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.

```
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:

```
mpixlc -o test-c test.c -L/soft/perftools/hpctw/lib –lmpihpm -L/
  bgsys/drivers/ppcfloor/bgpm/lib –lbgpm

mpixlf90 -o test-f test.f90 -L/soft/perftools/hpctw/lib –lmpihpm -
  L/bgsys/drivers/ppcfloor/bgpm/lib –lbgpm
```

  - 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>.
cores in use = 16, active threads per core = 4.
=================================================================
-----------------------------------------------------------------
mpiAll, call count = 1, avg cycles = 61829628876, max cycles = 61835872069 :
  -- Counter values summed over processes on this node ----
0      16279923298   Committed Load Misses
0     146867542314   Committed Cacheable Loads
0      15953740380   L1p miss
0     476269353705   All XU Instruction Completions
0     224066635616   All AXU Instruction Completions
0     444560789965   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>:
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 %
                  L1P buffer =   0.22 %
                  L2 cache   =  10.86 %
                  DDR        =   0.00 %
DDR traffic for the node: ld = 0.000, st = 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

- Installed in: /soft/perftools/mpiP
- Documentation at:
  - *www.alcf.anl.gov/user-guides/mpip*
  - *mpip.sourceforge.net/*

mpiP

- Using mpiP:
  - Recompile the code to include the mpiP library

```
mpixlc -o test-c test.c -L/soft/perftools/mpiP/lib/ -lmpiP -L /bgsys/drivers/
ppcfloor/gnu-linux/powerpc64-unknown-linux-gnu/powerpc64-bgq-linux/ -lbfd -
liberty

mpixlf90 -o test-f90 test.f90 -L/soft/perftools/mpiP/lib/ -lmpiP -L/bgsys/
drivers/ppcfloor/gnu-linux/powerpc64-unknown-linux-gnu/powerpc64-bgq-linux/ -
lbfd –liberty
```

  - 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

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

gprof

- 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: by default will generate one gmon.out for the 1st 32 ranks. Control ranks creating output by setting env variable:
    - BG_GMON_RANK_SUBSET = min:max:stride (eg. = 100:200:10)
  - 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
- Documentation: *www.alcf.anl.gov/user-guides/gprof-profiling-tools*

Scalasca

- Designed for scalable performance analysis of large scale parallel applications
- Instruments code and can collects 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
- Documentation:
  - *www.alcf.anl.gov/user-guides/scalasca*
  - *www.scalasca.org/download/documentation/documentation.html*

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
- Documentation:
  - *www.alcf.anl.gov/user-guides/openspeedshop*
  - *www.openspeedshop.org/wp/documentation/*

Darshan

- Darshan is a library that collects information on a programs IO operations and performance
- Darshan is enabled by default
- Upon successful job completion Darshan data is written to a file named

  <USERNAME>_<BINARY_NAME>_<COBALT_JOB_ID>_<DATE>.darshan.gz

  in the directory:
  - /gpfs/mira-fs0/logs/darshan/mira/<year>/<month>/<day>
- Documentation: *www.alcf.anl.gov/user-guides/darshan*

Steps in looking at performance

- Time based profile with at least one of the following:
  - Rice HPCToolkit
  - TAU
  - gprof
- MPI profile with:
  - HPCTW MPI Profiling Library
  - TAU
  - mpiP
- Gather performance counter information for critical routines:
  - HPCTW
  - PAPI
  - BGPM

Code to test with

- Relatively simple C code for testing tools is available on Vesta in the directory:
  - /home/sparker/tmp/atpesc
- Consists of a main routine that calls a number of small kernels inside a loop: matrix multiple, matrix vector, streaming vector update, random table access, point-to-point, allRecude, bcast
- Build by running “make all”
- Built in instrumentation when compiled with “-DINST”, can be used to compare with and understand tool output
- Brief description of call sequence in “main.c”