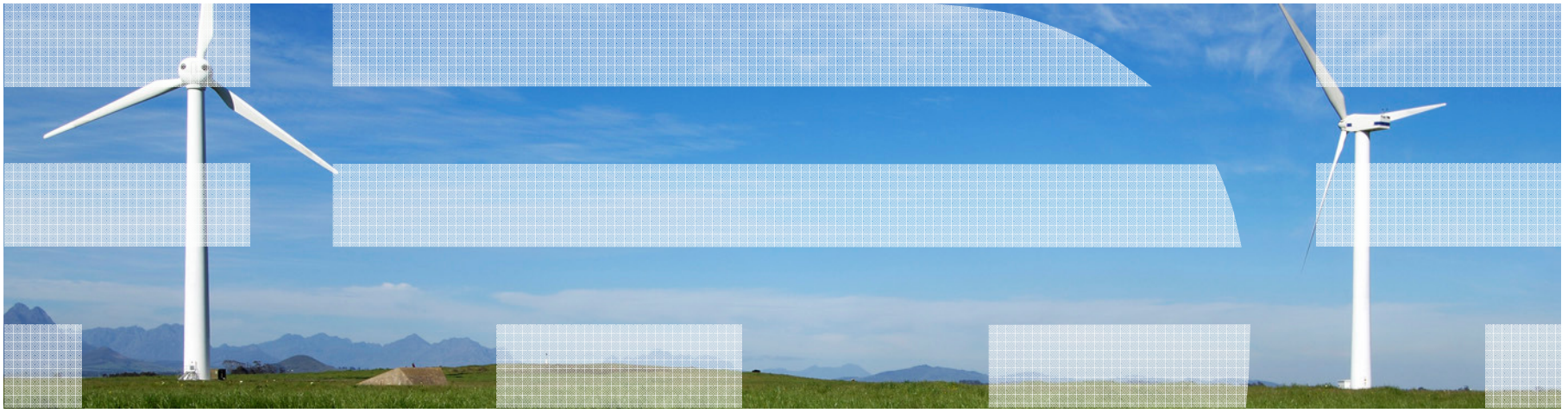
Blue Gene/Q User Workshop

Performance analysis



Agenda

- Code Profiling – Linux tools
  - GNU Profiler (Gprof)
  - bfdprof
- Hardware Performance counter Monitors
- IBM Blue Gene/Q performances tools
  - Internal mpitrace Library
  - IBM HPC toolkit
- Major Open-Source Tools
  - SCALASCA (fully ported and developed on BG/Q – Juelich Germany)
  - TAU

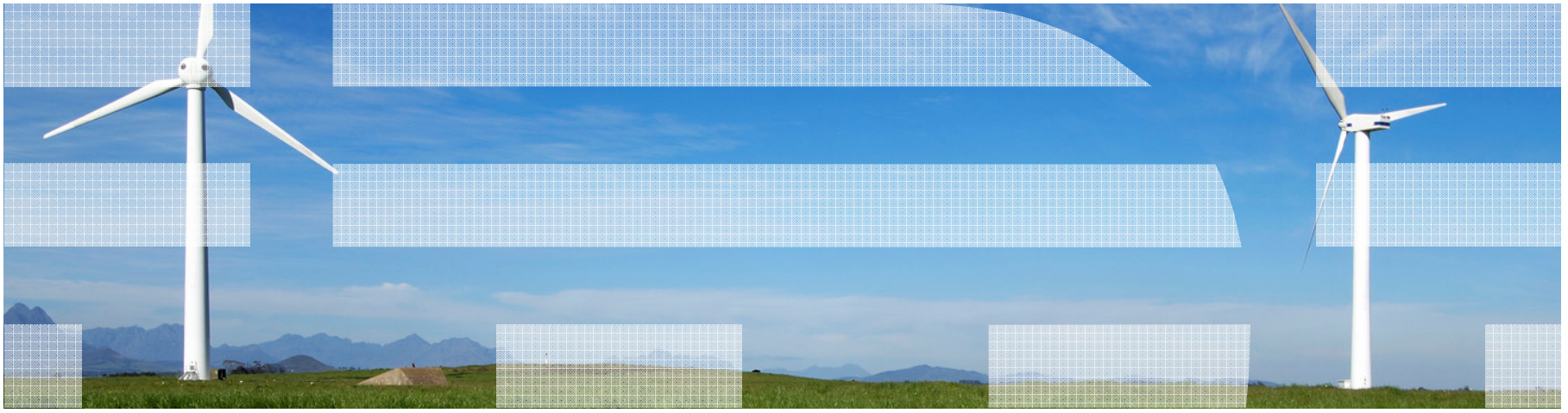
- IBM System Blue Gene/Q Specifics
  - Personality

Using XL compiler wrappers

- **Tracing functions in your code**

  – Writing tracing functions – example in XL Optimization and Programming guide
    - __func_trace_enter is the entry point tracing function.
    - __func_trace_exit is the exit point tracing function.
    - __func_trace_catch is the catch tracing function.
  – Specifying which functions to trace with the **-qfunctrace** option.

Standard code profiling



Code profiling

- Purpose
  - Identify most-consuming routines of a binary
    - In order to determine where the optimization effort has to take place
- Standard Features
  - Construct a display of the functions within an application
  - Help users identify functions that are the most CPU-intensive
  - Charge execution time to source lines
- Methods & Tools
  - GNU Profiler, Visual profiler, addr2line linux command, …
    - new profilers mainly based on Binary File Descriptor library and **opcodes** library to assemble and disassemble machine instructions
    - Need to compiler with **-g**
  - Hardware counters
- Notes
  - Profiling can be used to profile both serial and parallel applications
  - Based on sampling (support from both compiler and kernel)

GNU Profiler (Gprof) | How-to | Collection

- Compile the program with options: **-g –qfullpath + -pg** (for gno profiler)
  - Will create symbols required for debugging / profiling
- Execute the program
  - Standard way
- Execution generates profiling files in execution directory
  - **gmon.out.<MPI Rank>**
    - Binary files, not readable
  - Necessary to control number of files to reduce overhead
- Two options for output files interpretation
  - GNU Profiler (Command-line utility): **gprof**
    - **gprof <Binary> gmon.out.<MPI Rank> > gprof.out.<MPI Rank>**
  - Graphical utility / Part of HPC Toolkit GUI: **Xprof**
- Advantages of profiler based on Binary File Descriptor versus gprof
  - Recompilation not necessary (linking only)
  - Performance overhead significantly lower

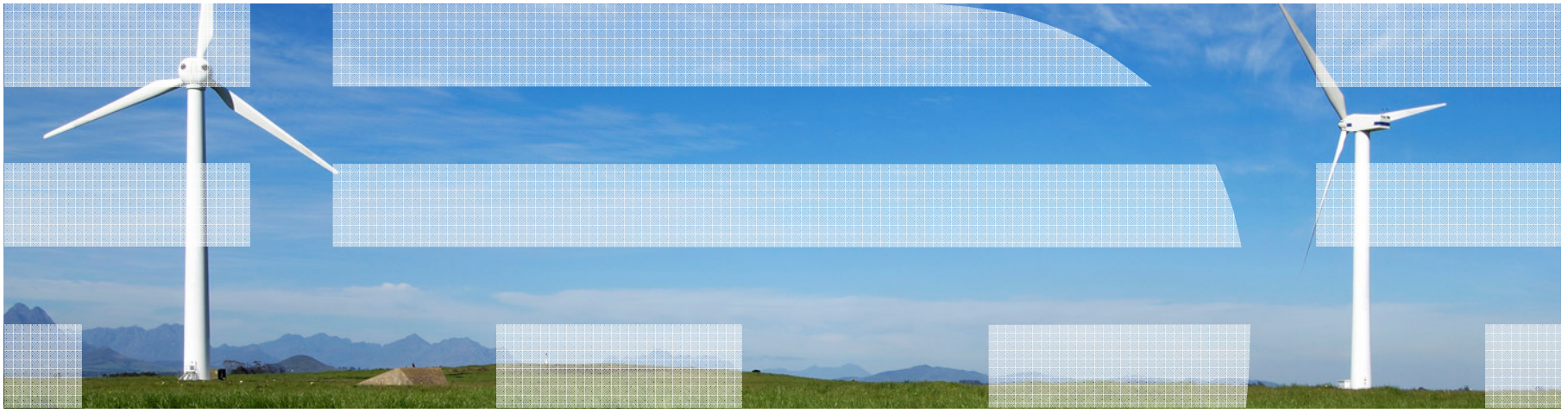
Using GNU profiling

/bgsys/drivers/ppcfloor/gnu-linux/bin/**powerpc64-bgq-linux-gprof**

- **BG_GMON_RANK_SUBSET=N** /* Only generate the gmon.out file for rank N. */
- **BG_GMON_RANK_SUBSET=N:M** /* Generate gmon.out files for all ranks from N to M. */
- **BG_GMON_RANK_SUBSET=N:M:S** /* Generate gmon.out files for all ranks from N to M. Skip S; 0:16:8 generates gmon.out.0, gmon.out.8, gmon.out.16 */

- The base GNU toolchain does not provide support for profiling on threads
- Profiling threads
  - **BG_GMON_START_THREAD_TIMERS**
    - Set this environment variable to “all” to enable the SIGPROF timer on all threads created with the pthread_create() function.
    - “nocomm” to enable the SIGPROF timer on all threads except the extra threads that are created to support MPI.
  - Add a call to the gmon_start_all_thread_timers() function to the program, from the main thread
  - Add a call to the gmon_thread_timer(int start) function from the thread to be profiled
    - 1 to start, 0 to stop

Hardware performance monitors



Hardware Counters

- Definition
  - Extra logic inserted in the processor to count specific events
  - Updated at every cycle
  - Strengths
    - Non-intrusive
    - Very accurate
    - Low overhead
  - Weakness
    - Provides only hard counts
    - Specific for each processor
    - Access is not well documented
    - Lack of standard and documentation on what is counted

    => **useful to use a higher level software**
- Purpose of a high level software (like IBM HPM)
  - Provides comprehensive reports of events that are critical to performance on IBM systems
  - Gathers critical hardware performance metrics
    - Number of misses on all cache levels
    - Number of floating point instructions executed
    - Number of instruction loads that cause TLB misses
  - Helps to identify and eliminate performance bottlenecks

BG/P versus BG/P Hardware Counters

- BG/P
  - 256 64bit counters on Blue Gene/P
    - 72 of these counters are core specific while 184 counters are shared across the four PowerPC 450 cores
    - Max 4t → 288 independent core counts per process
    - shared counters measure events related to L2 cache, memory and network
  - Mode 0: cores 0 & 1
  - Mode 1: cores 2 & 3
- BG/Q
  - Much more complex
  - Collects data from all cores, L1P Units, L2, Message Unit, IO Unit, CNK Unit (virtual)
  - 600 events (414 core specific)
  - 24 counters are available per core
  - Can handle hardware threads
    - Can provide per-thread counts of processor events
    - But the 24 counters must be shared between threads
    - 4 Hw Threads → 6 counters per thread
    - Max 64t → 384 independent core counts per process
  - Supports multiplexing
    - Provides ability to count more than the set (24) number of events
    - Basic Idea: Start with one set of events, after a time interval, set another event set

Multiplexing

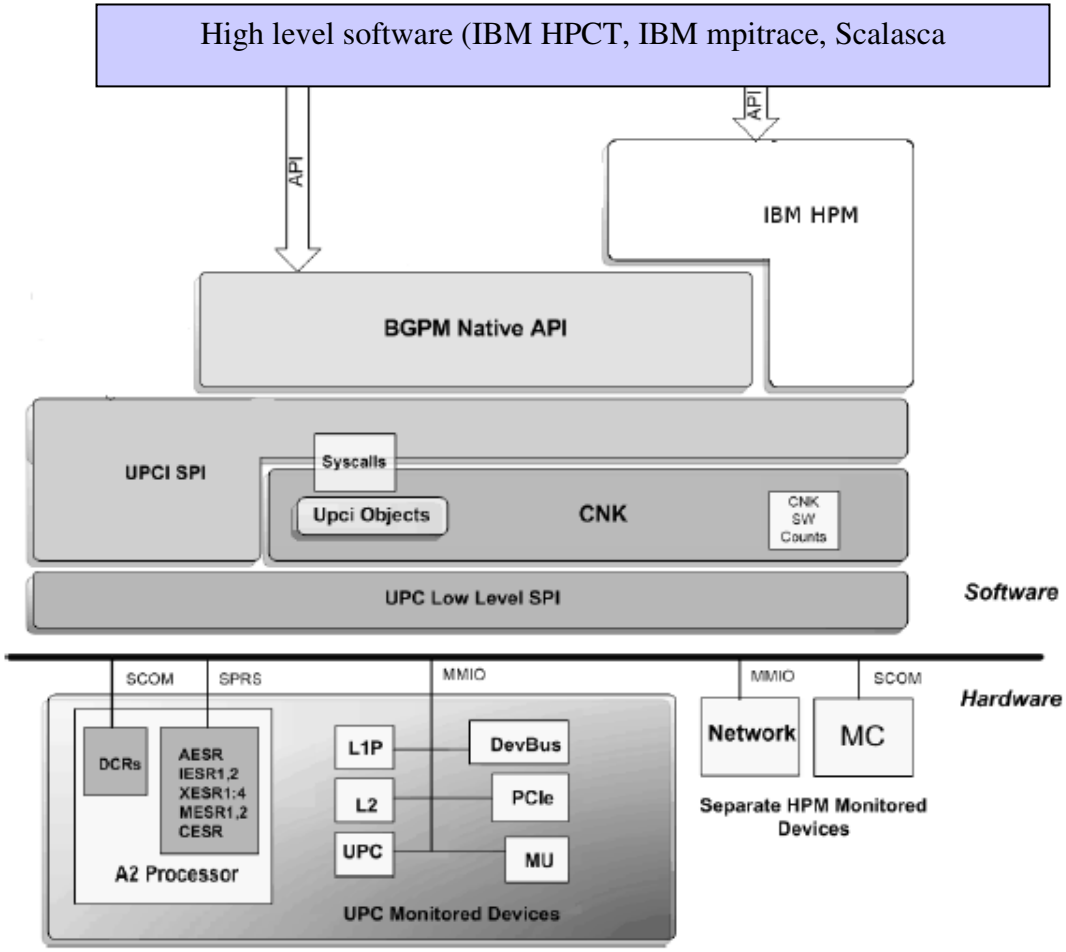
- Provides ability to count more than the set (24) number of events
- Basic Idea: Start with one set of events, after a time interval, set another event set
  - Counter architecture identifies conflicts
  - Saves counts of conflicted events
  - Clears the counters and sets them to count new event
  - After another time interval switches back to original
- **Advantage**: Can collect a lot more data in a single run
- **Disadvantage**: Multiplexed counter accuracy is comprimsed
  - The counts are not correct unless the windows equally cover the code.
  - One set may only register events from one part of the algorithm
  - You cannot add/compare counts from events in the different groups
- Use to get general overview of the counter values to see if they should be investigated in more detail

Nomenclature

- UPC
  - Universal Performance Counting
    - Hardware and low-level software
- **BGPM**
  - **Blue-Gene Performance Monitor**
    - Mid-Level software providing access to counters
- HPM from IBM HPC toolkit
  - Hardware Performance Monitor
    - High-Level software providing access to counters (for devs)

- Counter types
  - AXU, QPX, QFPU
    - All refer to the Quad FP Unit
  - XU, FXU
    - The Execution Unit (Fixed-Point Unit)
    - In PAPI FXU means floating-point unit!
  - IU
    - The instruction unit (Front-End of pipeline)

BG/Q Counter Related Software Layers



Performance Application Programming Interface (PAPI)

- PAPI-C library - performance application programming interface (PAPI)
  - http://icl.cs.utk.edu/papi

- The PAPI-C features that can be used for the Blue Gene/Q system include:
  - A standard instrumentation API that can be used by other tools.
  - A collection of standard preset events, including some events that are derived from a collection of events. The BGPM API native events can also be used through the PAPI-C interfaces.
  - Support for both a C and a Fortran instrumentation interface.
  - Support for separate components for each of the BGPM API unit types:
    - Punit counter is the default PAPI-C component.
    - L2, I/O, Network, and CNK units require separate component instances in the PAPI-C interface.
  - See PAPI and BGPM docs for which BGPM events map to PAPI events

BGPM (Blue-Gene Performance Monitor) | Details

- BGPM API functions to program, control, and access counters and events from the four integrated hardware units and the CNK software counters.
- Doxygen documentation gives detailed information on BGPM and counter architecture
  - ***/bgsys/drivers/ppcfloor/bgpm/docs/html/index.html***
- 4 main collection sources
  - Processor (Punit)
    - 24 Counters. Thread Aware. Multiple units e.g. Load-Store, Floating-Point, L1p ..
  - L2
    - 6 counters per slice. Not thread/core aware
    - Usuallly operate in combined mode
  - IO Unit (MU, PCIE, DevBus)
    - Counts static set of events. Not thread/core aware
  - Network Unit
    - 6 counters per link (10 torus links, 1 I/O link)
    - Each link can only be counted by a single thread
- 3 major modes of operation:
  - Software distributed mode
    - Each software thread configures and controls its own Punit counters
  - Hardware distributed mode
    - A single software thread can configure and simultaneously control all Punit counters for all cores
  - Low latency mode
    - Provides faster start and stop access to to the Punit counters

BGPM | Events, Instructions, OpCodes

- Instructions
  - Either XU or AXU depending on which pipeline they pass through
  - Instructions can be microcoded – Made up of 2+ ucode sub-operations
  - Total instructions = Non-Ucoded + Ucoded + Ucoded sub-ops
  - Events couting instructions can
    - Count only non-microcoded instructions (1 instruction == 1 operation)
    - Count microcoded instructions – but not the sub-operations
    - Count suboperations only
    - Various combinations of the above
- OpCodes v Unit Events
  - The opcode counter counts completed operations – looking at the end of the pipeline
  - The unit events are counted by the units themselves – internal
  - OpCode counter can discrimate sub-ops → provide counts equivalent to instructions
- Instructions and Opcodes are associated with the pipeline
  - Events couting them come from IU, XU, AXU, Opcode counter
- Events in the other units (LSU,MMU,L1P) are not directly pipeline related
  - Result of instructions in the pipeline
  - e.g. Load instructions go through XU pipeline and then are dispatched to LSU
- Events can be divided into three main groups based on how they realted to processor cycles
  - Cycle Only Events e.g. Number of cycles pipeline stalled
  - Single cycle events – events and cycles are synonymous.
    - Instruction and opcode counting
  - Multi-cycle events – Only can count the occurances of the event – no cycle information

BGPM | Processor Unit Counters

- Processor has a number of sources
  - Instruction Unit (IU) – 35 events
  - Floating Point Unit (AXU) – 9 events
  - Execution Unit (XU) – 35 events
  - Load-Store Unit (LSU) – 32 events
  - Memory-Managment Unit (MMU) – 31 events
  - L1P – 66 events
  - Wake Up Unit (WU) – 2 events
  - Opcode Counter – Counts operations by related “groups”
    - 24 XU groups, 25 AXU groups
    - 6 AXU FLOP groups - > since 1 op → multiple flops
    - 6 AXU Inst groups (giving the instructions counts related to above)
- The main units (IU, AXU, XU, LSU, MMU) can track max 8 events (4 threads → 2 per thread)
- When counting unit events the 24 counter are hardware thread specific (software distributed)
  - Each thread can only count max 12 unit events!
  - Due to wiring/hardware considerations
- However the OpCode counter can use all 24 counters.
- L1P unit is most complicated in terms of what can/can't be counted at the same time
  - Because it does prefectching plus interfaces between L2 and core
  - 4 modes – list, stream, base, switch (requests to crossbar)

BGPM | Tips

- Check the BGPM Tips Page!
  - Docs/bgpm_event_tips.html
- Gives a detailed mapping of cycles to events
  - how the total number of cycles can be broken into different events
  - e.g. Total Cycles - IU Issues + IU Stall = IU Empty
    - PEVT_CYCLES - PEVT_IU_TOT_ISSUE_COUNT + PEVT_IU_IS1_STALL_CYC = IU Empty
- Gives a (fairly) detailed explanation of the pipeline/event relationships
  - Look at this in more detail tomorrow

BGPM | HPM (IBM High Performance Monitoring) part of HPC Toolkit

- **HPM Principle**
  - IBM HPM Library provides a very easy use of HPM
    - Allows access to most relevant hardware counters
  - Provides pre-set groups of events that can be counted together
    - And counter on a per-hardware thread basis
  - Handles multiplexing for groups that require it
  - Handles overflows and other counter related issues
  - Outputs data in easy to read file-format
  - Can collect data on multiple parts of a code simultaneously

- HPM provides “readable” names for BGPM events, However it does not tell you the underlying BGPM name (see HPCT docs for map)

- **Default group detail**:
  - Total Loads - PEVT_LSU_COMMIT_LD_CACHABLE_LOADS
  - XU Instructions - PEVT_INST_XU_ALL
  - AXU Instructions - PEVT_INST_QFPU_ALL
  - L1 Data Cache Miss - PEVT_LSU_COMMIT_LD_MISSES
  - L1P Misses - PEVT_L1P_BAS_MISSES
  - FLOPS - PEVT_INST_QFPU_FPGRP1_INSTR
  - Total Cycles - PEVT_CYCLES

- The readable names are closely related to the description strings of the events which helps

HPM | How-To

- Add hpmInit & hpmTerminate statements to code
  - Directly after/before MPI Init/Finalize
  - Need to add header files (hpm.h, f_hpm.h)
- Bracket routines to be profiled with hpmStart(name) and hpmStop(name)
  - You can nest calls
- Link with IBM HPM Library – libhpc or libhpc_r (for threaded)
- Execute with following environment variables
  - HPM_EVENT_GROUP=
- Execution produces one HPM file per MPI task
  - hpmCount_(Process Id).*
- HPM Environment Variables
  - HPM_OUTPUT_PROCES (all/root)
  - HPM_SCOPE (process|node)
  - HPM_ASC_OUTPUT=yes (write output filed for peekperf)
  - HPM_METRICS (yes|no) – Print derived metrics
  - HPM_EXCLUSIVE (yes|no) – Outer nested regions counted separately to inner

HPM | Groups

- All HPM output gives total cycles between start/end profiled region
- Groups – Each HPM group provides different set of processor events
  - Default (-1) – See below - Non-multiplexed
  - 0: Instructions & LSU events
  - 1: Branch-Prediction
  - 2: Floating-Point breakdown
  - 3: Large mix of counters
  - 4: L1 Stream Prefetching
  - 5: Pipelining
- Default Group (Per Thread)
  - Total Loads
  - Total XU Instructions
  - Total AXU Instructions
  - L1 Data Cache Misses
  - L1P Misses
  - FLOPS
  - MFLOPS (derived)
- All Groups Provide
  - L2 Hits
  - L2 Misses
  - L2 Lines Loaded From Main Memory
  - L2 Lines Stored To Main Memory

HPM | Derived Metrics

- Total Instructions – Total XU + Total AXU
  - Can calculate instruction mix from this
  - Non-Load Instructions (Total XU - Loads)
- Throughput
  - Ins/Cycle (Total Inst/Total Cycles)
  - % Max issue rate (Ins/Cycle)/2.0
- L1 Hit %: (Loads – L1 Misses)/Loads
- L1P Hit %: (L1Misses – L1PMisses)/Loads
- L2 Hit %: (L1PMisses – L2Misses)/Loads
  - Not cannot use L2 Hits due to prefetch engine
- RAM Hit %: (L2 Misses)/Loads
- RAM Traffic: (L2 Lines Stored/Loaded)*128/Total Cycles
  - Max traffic is 13 Bytes/Cycle (average).

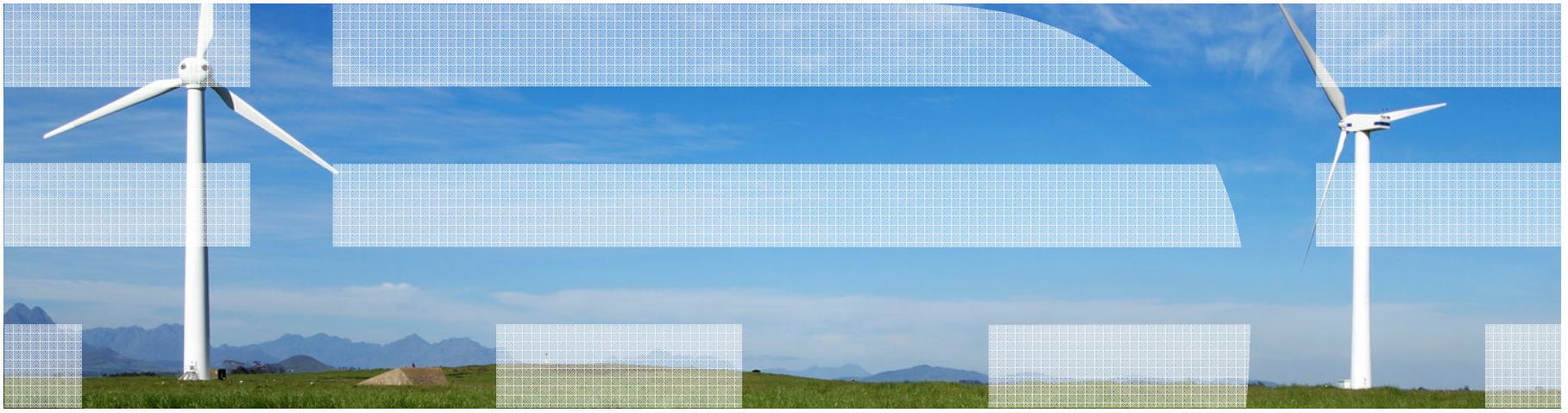
Hardware Counters (HC) | Tips

- Counters tell you exactly what code is doing
  - Don't tell you if what its doing is sub-optimal
  - You might find the reason for poor performance – and that you can't do anything about it
- Two ways in which you can have issues:
  - Compiler or hardware details make the algorithm act differently than expected
    - HC can help identify this
  - You have non-optimal implementation of the algorithm
    - More difficult → HC can possible point you in right direction
- **Need to have a good idea of what you expect the algorithm to be doing**
- Usually have a base set of operations that is being iterated many times
  - Work out details of this base set
    - FLOPS
    - Misses (Min/Max) – i.e. expected loads from various levels
    - Code Balance: (Floating Point Instructions/Ops)/Loads (choose level)
- Divide counters by number of iterations of base calculation
  - Makes counters more understandable

Hardware Counters (HC) | Tips

- Check instruction mix:
  - Is there anyway you can lessen the non-FP instructions?
    - Eliminate loads (from any level)
    - Reduce pointer related calculations
  - Less instructions → less cycles → better performance
- Check misses
  - Are there more misses/hits to lower cache levels than you expect?
  - Compare actual v expected code-balance
- Calculate average latency per load
  - (Cycles – Instructions)/Loads
  - Is it higher than expected?
  - Compare to (L1Hit%)*L1Lat + (L2Hit%)*L2Lat etc.
- The above two measures can indicate punit resource contention e.g.
  - L1 Cache lines (Cache Thrashing)
  - Load/Store queue
  - L1P contention
- With hardware threads compare 1 thread to 4 threads to see changes
  - Also 4t on one core to 4t on separate cores
- **Note**: L2 Hits is misleading due to the L1P

IBM MPI communications tracing library (mpitrace)



IBM MPI Trace library | Principles

- MPI wrappers for BlueGene have a number of optional features that can be controlled by setting environment variables
- Capabilities:
  - ***MPI Trace Features***
  - ***Collects all MPI communications of an application***
  - ***Measures time spent in the MPI routines***
  - ***Provides call graph for communication subroutines***
  - ***Collective imbalacing***
  - ***BG/Q infos: block shape, task and IO bridge coordinates, # of IO nodes***
  - ***Access to hardware counters***
  - ***Code Profiling (gprof and bfdprof), including hardware counters***
  - ***Posix IO traces***

- There is one combined wrapper-set for apps that use Fortran and C:
  - **libmpitrace.a** : wrappers for MPI
  - **libmpihpm.a** : wrappers for MPI + hardware counters for pure MPI applications
  - **libmpihpm_smp.a** : wrappers for MPI + hardware counters for mixed MPI + OpenMP applications

  To enable IO traces
  - **libmpitraceio.a** : wrappers for MPI and IO only
  - **libmpihpmio.a** : wrappers for MPI + IO + hardware counters for pure MPI applications
  - **libmpihpm_smpio.a** : wrappers for MPI + IO + hardware counters for mixed MPI + OpenMP applications

- **IBM HPC Took provides similar functions with more features (openmp, output control, code sections, …), customable + graphic interface, *(different for IO traces), but with some limitations – use most of the same env variables***

IBM MPI Trace library | Principles

- Usage – compile code with **–g** : allows translation from instruction address to source-file and line number
  - Link with library for the IO version
    - -Wl,-wrap,open -Wl,-wrap,close -Wl,-wrap,read -Wl,-wrap,write -Wl,-wrap,fopen -Wl,-wrap,fclose -Wl,-wrap,fread -Wl,-wrap,fwrite
      <Install Directory>/libmpitrace.a
  - for **libmpihpm.a** and **libmpihpm_smp.a** add in the link
    - /bgsys/drivers/ppcfloor/bgpm/lib/libbgpm.a
      /bgsys/drivers/ppcfloor/spi/lib/libSPI_upci_cnk.a

- Output Files
  - mpi.profile.<Process ID>.#rank
  - hpm_process_summary. .<Process ID >.#rank
  - hpm_job_summary. .<Process ID >.#rank
  - Pattern. .<Process ID>.#rank
  - events.trc
  - Gmon.out or vmon.out .#rank

  - To avoid <Process ID> suffix : export TRACE_OMIT_JOBID=yes

IBM MPI Trace library | Sample output

```
-----------------------------------------------------------------
MPI Routine                    #calls     avg. bytes      time(sec)
-----------------------------------------------------------------
MPI_Comm_size                       1            0.0         0.000
MPI_Comm_rank                       1            0.0         0.000
MPI_Isend                        5738         2398.6         0.050
MPI_Irecv                        2163         2738.7         0.010
MPI_Waitall                      1919            0.0         0.028
MPI_Reduce                          3            8.0         0.000
-----------------------------------------------------------------
total communication time = 0.087 seconds.
total elapsed time       = 3.922 seconds.
user cpu time            = 3.890 seconds.
system time              = 0.030 seconds.
maximum memory size      = 30012 KBytes.

-----------------------------------------------------------------
Message size distributions:

MPI_Isend                  #calls     avg. bytes      time(sec)
                             2389            8.0         0.012
                             3349         4104.0         0.038

MPI_Irecv                  #calls     avg. bytes      time(sec)
                              721            8.0         0.001
                             1442         4104.0         0.008

MPI_Reduce                 #calls     avg. bytes      time(sec)
                                3            8.0         0.000
```

IBM MPI Trace library

- By default trace from MPI_Init to MPI_Finalize
- Controlling by region

MPI

– C example:

```
summary_start();
  do_work();
summary_stop();
```

Fortran

```
call summary_start()

call summary_stop()
```

CPU Profiling

vprof_start(), vprof_stop()

Hardware-counters

HPM_Start("timesteps"), HPM_Stop("timesteps");

IO

jio_start(), jio_stop()

Event-tracing

trace_start(), trace_stop()

Mpitrace env variables

- **Controling output**
  - export PROFILE_ACTIVE=no
  - export SAVE_ALL_TASKS=yes
  - export SAVE_LIST=0,2,4,6,8,10, TRACE_MAX_RANK
  - export TRACE_DIR=/path/to/your/profile/files
  - Export TRACE_DISABLE_LIST=

- Export TRACE_SEND_PATTERN=yes
  - for each message sent, the MPI wrappers will identify the source and destination torus coordinates, and keep track of the total number of byte-hops for each destination rank.

- **IO**
  - export PROFILE_JIO=yes
    - JIO_LEVEL=SUMMARY
    - JIO_LEVEL=DETAILED
    - JIO_LEVEL=TRACE - traces *every* I/O. Don't use it :-) traces in stderr

- **CPU profiling**
  - Gmon profiling: Refer to documentation for gmon control on BG/Q
  - vmon profiling
  - export VPROF_PROFILE=yes
  - cprof or bfdprof command your.exe vmon.out.n > cprofile_n.txt &
    - Profile tips per file, function and code annotations

Some mpitrace env variables

- **Event tracing**
  - TRACE_ALL_EVENTS=yes
- **MPI collective imbalacing**
  - export PROFILE_IMBALANCE=yes
  - export PROFILE_IMBALANCE_MPIO=yes
- **Hardware counter**
  - Can change group using HPM_GROUP
  - Can change scope to per_process using HPM_SCOPE

  - HPM_SCOPE=[node, process, thread]
  - HPM_MASK=10000000 | 01111111 ; master thread | all except master thread
  - HPM_GROUP=2
    - count all FPU-related instructions
  - HPM_GROUP=5
    - All possible integer/load/store instructions

Profiling Basics

Can do profiling via timer interrupts : -pg or user-callable profil() routine.
Interrupts occur at 100 per sec … so each hit corresponds to 0.01 sec.

Can also do profiling with hardware counters … best bet is with A2 events.
Set the BGPM event, and set the threshold value … get an interrupt when the counter increments by the threshold value, trap the address, build a histogram.

The basic profiling data is a histogram of “hits” as a function of instruction address.

bfdprof your.exe vmon.out.N > profile.N (typical use)

bfdprof example : gyro BGPM event = cycle counter

Got a total of 156543 hits at 7476 program-counter locations.
HPM sampling using event = 211, threshold = 1600000.

1000 samples/sec

```
############################
Function-level profile:
############################
    tics   function-name
-------------------------
   37843  gyro_field_interpolation$$OL$$2
   32739  gyro_operators_on_h$$OL$$1
   15009  gyro_nl_direct$$OL$$1
    7944  zmv4vfe
    5736  gyro_moments_plot$$OL$$1
    5113  zlnrsvfa
    4726  gyro_field_interpolation$$OL$$1
    4691  gyro_velocity_sum$$OL$$1
    4640  zunrsvfa
    2894  gyro_rhs_total$$OL$$2
    2800  gyro_get_delta_he$$OL$$1
    2588  gyro_collision_kernel$$OL$$2
    2541  gyro_timestep_implicit
    2170  blend_f
    1520  gyro_get_he_implicit$$OL$$1
    1308  gyro_tau_derivative$$OL$$1
```

OpenMP codes have different functions for each parallel region.

bfdprof example : gyro BGPM event = cycle counter

```
##########################
Source-file profile:
##########################
    tics   source-file
-------------------------
   42587  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_field_interpolation.f90
   32744  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_operators_on_h.f90
   23022  unknown
   15021  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_nl_direct.f90
    5827  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_moments_plot.f90
    4730  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_velocity_sum.f90
    3127  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_rhs_total.f90
    2994  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_collision_kernel.f90
    2800  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_get_delta_he.f90
    2541  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_timestep_implicit.f90
    2170  /gpfs/bgq0/walkup/projects/gacode/gyro/BLEND/BLEND_F.f90
    1521  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_get_he_implicit.f90
    1308  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_tau_derivative.f90
    1184  /gpfs/bgq0/walkup/projects/gacode/gyro/BLEND/BLEND_f3.f90
    1089  /gpfs/bgq0/walkup/projects/gacode/gyro/src/gyro_g_squared.f90
```

Functions compiled without -g have “unknown” source file. Typical examples are math library routines. This app uses ESSL.

bfdprof example : gyro BGPM event = cycle counter

```
############################
Annotated source for file: /gpfs/…/gyro_operators_on_h.f90
############################
  tics | source
       | …
       |   if (n_field < 3) then
       |
       |     do i = ibeg, iend
     1 |       do m=1,n_stack
    28 |         temp = (0.0,0.0)
    36 |         do i_diff=-m_gyro,m_gyro-i_gyro
 32129 |           temp = temp + w_gyro(m,i_diff,i,p_nek_loc,is)*hh(m,i+i_diff)
   192 |         enddo ! i_diff
     5 |         gyro_h(m,i,p_nek_loc,is) = temp
    25 |       enddo ! m
    33 |     enddo ! i
```

Inner loop over i_diff has bad stride. Can re-order the loops, make the “m” loop innermost … get stride-1 but will have more load/store instructions.

bfdprof example : gyro BGPM event = L1P misses

```
Got a total of 22962 hits at 1159 program-counter locations.
HPM sampling using event = 146, threshold = 30000.

#########################
Function-level profile:
#########################
    tics   function-name
-------------------------
   12217  gyro_operators_on_h$$OL$$1
    1549  gyro_moments_plot$$OL$$1
    1422  zmv4vfe
    1201  gyro_field_interpolation$$OL$$1
    1013  gyro_nl_direct$$OL$$1
     860  zunrsvfa
     562  zlnrsvfa
     387  gyro_field_interpolation$$OL$$2
     318  gyro_g_squared$$OL$$1
     305  gyro_tau_derivative$$OL$$1
     219  gyro_rhs_total$$OL$$2
     201  gyro_collision_kernel$$OL$$2
     195  gyro_conserve_number$$OL$$1
     183  gyro_nonlinear_flux$$OL$$1
     170  gyro_get_delta_he$$OL$$1
     124  MPIDO_Barrier
     117  gyro_collision_kernel$$OL$$1
```

Most of the L1P misses are in gyro_operators_on_h().

bfdprof example : gyro BGPM event = L1P misses

Original code:

```
#########################
Annotated source for file: /gpfs/…/gyro_operators_on_h.f90
#########################
  tics | source
       | …
       |   if (n_field < 3) then
       |
       |     do i = ibeg, iend
       |       do m=1,n_stack
       |         temp = (0.0,0.0)
       |         do i_diff=-m_gyro,m_gyro-i_gyro
  12208|           temp = temp + w_gyro(m,i_diff,i,p_nek_loc,is)*hh(m,i+i_diff)
       |         enddo ! i_diff
       |         gyro_h(m,i,p_nek_loc,is) = temp
       |       enddo ! m
      7|     enddo ! i
```

No surprise : L1P misses are due to bad stride in the “i_diff” loop.

bfdprof example : gyro BGPM event = L1P misses

Tuned code:

```
############################
Annotated source for file: /gpfs/…/gyro_operators_on_h.f90
############################
  tics | source
       | …
       |   if (n_field < 3) then
       |
       |     do i = ibeg, iend
       |       gyro_h(:,i,p_nek_loc,is) = (0.0,0.0)
       |       do i_diff=-m_gyro,m_gyro-i_gyro
       |         do m = 1, n_stack
    24 |           gyro_h(m,i,p_nek_loc,is) = gyro_h(m,i,p_nek_loc,is) &
       |             + w_gyro(m,i_diff,i,p_nek_loc,is)*hh(m,i+i_diff)
       |         end do ! m
       |       enddo ! i_diff
     2 |     enddo ! i
```

L1P misses are almost all gone. Time in this routine improved from 0.328 to 0.195 seconds per call … overall app impact ~10% improvement.