Internal Use MPI Wrappers for BGQ

Bob Walkup (walkup@us.ibm.com), Sept. 7, 2011

Quick Start.

(1) link your application with one of libmpitrace.a, or libmpihpm.a, or libmpihpm_smp.a
(2) run the code
(3) look at the text outputs that are produced, using the editor of your choice

Current snapshots for the libraries are located here:

bgqssn4.watson.ibm.com:/bgsys/home2/walkup/mpi_trace/bgq/lib

Use libmpitrace.a if you want just MPI performance data. If you want both MPI and hardware-counter data, then use one of either libmpihpm.a for pure MPI codes, or libmpihpm_smp.a for codes that use a mix of OpenMP and MPI. When using libraries with hardware-counter support, it is necessary to link with the basic counter libraries:

/bgsys/drivers/ppcfloor/bgpm/lib/libbgpm.a
/bgsys/drivers/ppcfloor/spi/lib/libSPI_upci_cnk.a

An example of the MPI timing summary for the Sequoia benchmark SPHOT is shown below.

```
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.
```

In this particular case the total amount of time spent in MPI is a small fraction of the elapsed time, and the parallel efficiency is very high.

Interpretation of MPI timing data requires some care. The most common issue is sorting out the effects of process synchronization and actual message transmission. To do this, it helps to have access to basic information, such as ping-pong times as a function of message size for processes that are either on the same node, or on different nodes. Similarly, basic information on the performance of collective MPI routines is useful. For example, if an application spent 100 seconds making 4000 calls to MPI_Allreduce using 8-byte messages, then that time must be essentially all process synchronization time, because if the processes were in sync, the network time required for one 8-byte MPI_Allreduce would be very roughly in the 10-100 microsecond range, depending on the communicator, reduction operator, etc. It is very common that the majority of the time spent in MPI is related to process synchronization instead of the latency or bandwidth characteristics of the network.

A sample makefile might look like this:

```
CC = bgxlc_r

MPI_INC = -I/bgsys/drivers/ppcfloor/comm/xl/include

MPI_LIB = -L/bgsys/drivers/ppcfloor/comm/xl/lib -lmpich -lmpl -lopa \
          -L/bgsys/drivers/ppcfloor/comm/sys/lib -lpami \
          -L/bgsys/drivers/ppcfloor/spi/lib -lSPI_cnk -lrt

TRACE = /path/to/libmpitrace.a

CFLAGS = -g -O3 -qsimd=noauto $(MPI_INC)

LDFLAGS = -g

OBJS = your list of .o files ...

your.exe : $(OBJS)
$(CC) $(LDFLAGS)  -o  your.exe  $(OBJS)  $(TRACE)  $(MPI_LIB)

%.o : %.c
$(CC)  -c  $(CFLAGS)  $<
```

If you directly use one of the compiler commands such as bgxlc_r for the linking step, then you must explicitly include the system's MPI libraries, and the MPI wrapper library must come just before the system's MPI libraries. The paths for system libraries could possibly change with drivers, and so adjustments for MPI_LIB may be necessary. If you use some form of convenience script for the linking step, such as mpixlc_r, the system's MPI libraries are automatically included, and so it would be sufficient to add $(TRACE) at the end of the list of items to link.

Introduction.

The MPI specification provides profiling entry points for MPI routines. This enables end users to replace the normal MPI entry-points with their own wrappers, which can be useful for performance analysis and sometimes for debugging purposes. For example, you can use a wrapper for MPI_Recv() that has this outline:

```
int MPI_Recv(void * rbuf, int count, MPI_Datatype type, int src, int tag,
                  MPI_Comm comm, MPI_Status status)
{
    int rc;
    time1 = get_timer();
    rc = PMPI_Recv(rbuf, count, type, src, tag, comm, status);
    time2 = get_timer();

   log_the_event(...);
   return rc;
}
```

With the approach sketched above, it is simple to keep track of the total amount of time spent in MPI routines, and there are many such tools that can do this. It is preferable if the overhead required to keep track of timing information can be kept to a minimum. This usually requires a trade-off because it often takes extra time to obtain more detailed information. With the MPI wrappers for BlueGene, options are set via environment variables that are picked up in the wrapper for MPI_Init() or MPI_Init_thread(), and timing data is written when the application calls MPI_Finalize(). It is possible to control the region of code that is sampled … more on that later … but it is crucial for the application to eventually call MPI_Finalize(), so that you can get the timing data that was collected.

On BlueGene systems, most (but not all) applications use MPI; and so the MPI entry points can be used to make other kinds of tools such as hardware counters and traditional Unix-based profilers more accessible and/or control-able in a parallel environment. For example, when profiling parallel applications with the -pg option, it is convenient to use the MPI entry points to limit profiler output to a few selected MPI ranks … one does not normally want to get a gmon.out file from each of say 200000 MPI ranks. Similarly, for hardware counters, it is possible to do some data reduction in the wrapper for MPI_Finalize() to provide some useful derived metrics, without requiring a post-processing step. The MPI profiling entry points make it convenient to collect a considerable amount of performance data from large-scale parallel applications.

For BlueGene/Q, there are three libraries at the present time, and all three are in a development state. Source code and libraries can be found on the Watson BGQ front-ends:

```
bgqssn4.watson.ibm.com:/bgsys/home2/walkup/mpi_trace/bgq/src
```

libmpitrace.a : basic MPI wrappers for profiling and tracing
libmpihpm.a : all of the features of libmpitrace.a plus hardware-counters for pure MPI aplications
libmpihpm_smp.a: same as libmpihpm.a, but for codes that use both OpenMP and MPI

Due to the implementation of hardware-counters on BGQ, it is preferable to have separate libraries to get hardware-counter data from applications that use pure MPI vs. a mix of OpenMP plus MPI. To use any one of these libraries, you need to include it in the linking step for the application. BlueGene systems use the GNU loader (ld command), which is sensitive to link order; and so to properly pick up the altered MPI entry points, the wrapper library must come **after** all of the user's .o files and libraries, and **before** the system's MPI library. This approach works well for statically-linked applications, but a different approach would be better for dynamically-linked executables. In particular it would be better to start with the system-provided MPI shared library, and re-package it with instrumented MPI entry-points. That way one would only need to set a library path at run-time … no linking step would be needed. This approach has been available on AIX for some time, where the MPI library is provided in shared-library format only. Pointers from the user community on how to do this for BlueGene systems would be welcomed.

Features.

The MPI wrappers for BlueGene have a number of optional features that can be controlled by setting environment variables. The default behavior is to simply collect a summary indicating the total amount of time spent in MPI routines, and cumulative data such as the total number of calls for each MPI function and a crude histogram of message-sizes (for routines where that applies). This data is intended to give you a rough picture of the breakdown of time in the application: computation vs. communication. However, it is important to keep in mind that the timing data includes all of the time spent in MPI routines, not just the time required to move data over the network. Time in spent in MPI is frequently "wait" time, where one process has finished it's computational phase, and is waiting in an MPI routine to receive data from another process. It is expected that the default options for the MPI wrappers would be sufficient for most uses, but one can obtain additional data at the expense of some extra overhead. For example, an application might spend a large amount of time in MPI _Wait() … but there may be many such calls … so how do you find out which calls are performance-sensitive? You can set an environment variable, PROFILE_BY_CALL_SITE=yes. That will make the MPI wrappers obtain the call-stack for every MPI function call. Then, when the program calls MPI_Finalize(), you get a breakdown of time per call-site in the code. That clearly adds overhead, but it can be very useful for identifying performance-critical sections of the code. The call-stack consists of instruction-addresses, and so you will need to use the -g option when you compile and link your code, in order to properly associate source-files and line-numbers with the instruction addresses.

An example of a call-site section in the mpi_profile text files is shown here:

```
-----------------------------------------------------------------
Profile by call site, traceback level = 0
-----------------------------------------------------------------
Use addr2line to map the address to source file and line number.
Ensure -g is used for compilation and linking.
-----------------------------------------------------------------

communication time = 66.358 sec, call site address = 0x010f625c
   MPI Routine                 #calls        time(sec)
   MPI_Waitall                2203906          66.358

communication time = 25.804 sec, call site address = 0x010f6974
   MPI Routine                 #calls        time(sec)
   MPI_Allreduce               226151          25.804

communication time = 18.498 sec, call site address = 0x010f619c
   MPI Routine                 #calls        time(sec)
   MPI_Wait                   1180478          18.498
  ...
```

The "communication time" is just the total elapsed time spent in the MPI routine(s) that are associated with the given call-site address. To locate the source-file and line number for one of these routines, you have to use the addr2line utility … check the man page for addr2line for details. An example would be :

```
addr2line  -e  your.exe  0x010f625c
```

The -g option is needed to allow translation from instruction address to source-file and line number. It is generally not necessary to use the BGQ-specific version, powerpc64-bgq-linux-addr2line, which is in the driver's gnu-linux/bin directory; the front-end version, /usr/bin/addr2line, is sufficient. Instead of translating addresses one at a time, you can translate them all in one shot:

```
grep  "site address"  mpi_profile.#.rank | awk '{print $10}' | addr2line  -e  your.exe
```

If you want to tag the elapsed time with a call-site higher up the call chain, then you have to set TRACEBACK_LEVEL to an appropriate value when you run the application. This may be necessary when the application has its own messaging layer, where MPI calls are limited to that layer instead of appearing directly in the application code.

BlueGene systems have a torus network, and so the locality of communication, measured by the average number of hops, can be of interest. For technical reasons it is simplest to check the locality for point-to-point messages that use some flavor of MPI send. If you set the env variable TRACE_SEND_PATTERN=yes, then 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.

Sometimes it is really beneficial to get a time-resolved picture of the communication and computational phases. This "event-tracing" capability is also supported, however one has to be careful to limit the total volume of trace data, to keep it manageable. Event tracing is described in a later section.

It is often very valuable to collect standard Unix-based profiling data, using either the -pg option or an equivalent method. The MPI wrappers for BlueGene have features to support that … see the section on Unix-based profiling.

Hardware-counters can provide very valuable insight into the performance characteristics of applications, and so hardware-counter access has been built into the MPI wrappers for BlueGene … see the section on hardware-counters.

Controlling the region of code that is profiled.

With the MPI wrappers, it is always convenient to collect timing data from MPI_Init() to MPI_Finalize() … but sometimes one needs to focus on the MPI communication in a specific section of code. To do that it is necessary to instrument the code with calls to start and stop the collection of MPI summary data:

C example:
```
summary_start();
do_work();
summary_stop();
```

Fortran example:
```
call summary_start()
call do_work()
call summary_stop()
```

The first call to summary_start() zeroes out any data collected up to that point in the code, and summary_stop() temporarily stops the collection of MPI timing data. The start/stop calls can be inside a loop … one will get the aggregate timing data for the code block(s) that are bracketed by the start/stop calls. The start/stop routines are implemented in C, and so to use them in a C++ setting, you need to specify extern "C" linkage:

```
// C++ declarations
extern "C" void summary_start(void);
extern "C" void summary_stop(void);
```

The extern "C" qualifier is not needed when the calls are added to C code, and Fortran does not require an interface specification. Note that similar extern "C" qualifiers are needed for other functions that are used to control various options provided by the MPI wrappers.

Controlling Output.

The MPI wrappers produce plain text output that contains cumulative performance data. The default is to save all data from MPI rank 0, and the ranks that had the minimum, median, and maximum times in MPI. That way one can get a pretty good idea about most applications, without generating a large number of files. The MPI data is in files with names:

```
mpi_profile.#.rank
```

where # is a unique number for each job. The file for MPI rank 0 is special … it contains a summary of data from all other MPI ranks. If you really want to save a separate output file for each MPI rank, you can set an env variable:

```
export SAVE_ALL_TASKS=yes.
```

You can also save data from a specific list of MPI ranks by setting another env variable like this:

```
export SAVE_LIST=0,2,4,6,8,10
```

which in this example will result in output from MPI ranks (in the MPI_COMM_WORLD communicator) of 0, 2, 4, 6, 8, 10.

These output methods apply to other types of output, including hardware-counter output, and Unix-based profiler outputs. All outputs are written in the wrapper for MPI_Finalize(), and so it is crucial that the application calls MPI_Finalize().

The MPI rank that spent the least amount of time in MPI is often of particular interest because that rank has often done the most work, and other MPI ranks must wait for that one before they can continue. As a result, the rank with the minimum time in MPI is less affected by synchronization time than the other MPI ranks. There tends to be an inverse correlation between time spent in MPI vs. time spent doing computation, and so the rank with the minimum time in MPI is a good candidate for Unix-based profiling.

Normally output will be written in the working directory for the application. Sometimes, applications use temporary working directories that are deleted upon job completion. In a case like that you probably want to send the profiling output to some other directory. You can do that by setting an env variable:

```
export TRACE_DIR=/path/to/your/profile/files
```

and then the mpi_profile.#.rank files should be written in the TRACE_DIR directory upon completion of MPI_Finalize().

Obtaining the memory footprint.

The MPI wrappers use an SPI routine, Kernel_GetMemorySize(), to retrieve the maximum amount of heap memory used by the application. This routine is particularly useful, so a brief description is provided here:

```
#include <spi/include/kernel/memory.h>
```

```
uint64_t heap;
Kernel_GetMemorySize(KERNEL_MEMSIZE_HEAPMAX, &heap);
```

This function is meant to be in-lined, and I suggest using powerpc64-bgq-linux-gcc for compilation. You would need include paths :

```
-I/bgsys/drivers/ppcfloor  -I/bgsys/drivers/ppcfloor/spi/include/kernel/cnk
```

There are other options for the memory-query routine, described in the memory.h header file. The MPI wrappers use the code above to obtain the maximum heap memory reported by each MPI rank, inside the wrapper for MPI_Finalize(). The values for all of the MPI ranks are collected in the mpi_profile file for rank 0.

Unix-based Profiling.

Traditional Unix-based profiling remains popular for a good reason: it can provide very useful insight into the computational aspects of an application. The simplest method is to use the -pg option, preferably along with -g. You have choices: you can compile with the options -g -pg and link with -g -pg, and you will get call-graph data along with function-level profiling data. That imposes a significant amount of overhead per function-call, because when you add -pg as a **compiler** option, the compiler inserts a call to a routine that tracks the call-stack and the number of calls, etc., for every compiled function. It is often preferable to add -g as a compiler option (not -pg), and then specify both -g and -pg when you **link**. That way you get all of the function-level (and statement-level) profiling data, without the overhead associated with collecting call-graph information. This second method uses only the interrupt at 100 times-per-second to check the position of the program counter, or instruction address. The basic profile data is then a histogram showing how many interrupts occurred for each instruction address in the program text section. The standard profiling mechanism uses a buffer of unsigned short integers (range 0 – 65535) to keep track of the number of hits in each histogram bin. Since timer interrupts come at 100 per second, it is generally safe to profile for up to 10 minutes (600 seconds) without worrying about buffer overflow. Most applications have time spent in a large number of places and can be profiled for longer time periods without overflow … but one should be aware that buffer overflow can occur if the measurement interval is long enough. Also, sampling at 100 interrupts per second is quite coarse given that instructions are zipping through the cores at rates of ~10^9 instructions per second … so the limited sampling statistics needs to be kept in mind when interpreting profile data.

You can control the region of code that is profiled using the moncontrol() routine. In C, it would look like this:

```
int main(int argc, char * arg[])
{
  moncontrol(0); // turn off profiling
  initialization_code();
 …
  moncontrol(1);  // turn on profiling
  do_work();
  moncontrol(0);  // turn off profiling
 ...
}
```

For Fortran applications, and XL compilers, you need to let the compiler know that the moncontrol() routine uses an argument passed by value, not by reference; so in Fortran the sketch above would look like this:

```
program main
call moncontrol(%val(0))  ! turn off profiling
call initialization()
 …
call moncontrol(%val(1))  ! turn on profiling
call do_work()
call moncontrol(%val(0)) ! turn off profiling
 …
end
```

The MPI wrappers have a call to the mondisable() routine to automatically control the generation of gmon.out files. Instead of getting one gmon.out file per process, you will normally get just the gmon.out files for MPI rank 0 and the ranks that had the min, max, and median times spent in MPI routines. The same output controls that apply to the generation of mpi_profile files also apply to gmon.out files, with the exception of TRACE_DIR … because the system software provides the file name gmon.out.rank in code that is not user-accessible.

The gmon.out files are normally processed by gprof, or powerpc64-bgq-linux-gprof, but there are other tools that could be used. On previous systems, we have had xprofiler, an X-windows based tool that adds source-file annotation … this may be available for

BlueGene/Q.

An alternative tool is available from SourceForge, called vprof (for visual profiler), originally from Sandia National Labs. From my perspective, the most useful component of vprof is the character-mode tool, cprof, which can be used to annotate all source-files with clock ticks, producing a single comprehensive text file with performance data pertaining to computation. The cprof tool was written to work with the basic histogram data obtained by profiling, but it uses a different file formant, not gmon.out. In essence, the file-format required for cprof is much more compact … only the addresses with >0 ticks are saved. The vprof/cprof package was written using the profil() routine, which uses the same interrupt mechanism as traditional Unix-based profiling via -pg. In the original vprof package, routines were provided to start and stop profiling … but the routine to stop profiling also wrote-out the histogram data. It is more convenient to separate those steps, so that the output-file filters provided by the MPI wrappers can be used to control profiler output. At the present time, the API to control profiling using the vprof/cprof format via the MPI wrappers is:

C example:

```
…
vprof_start();  // start profiling
do_work();
vprof_stop();  // stop profiling
```

Fortran example:

```
…
call vprof_start()  ! start profiling
call do_work()
call vprof_stop()  ! stop profiling
```

If you add vprof_start()/vprof_stop() calls to your code, the wrapper for MPI_Finalize() will write out any profile data that has been collected, in vmon.out format. You can use the cprof utility to analyze the output. The current snapshot of cprof for BGQ is in:

```
bgqssn4.watson.ibm.com:/bgsys/home2/walkup/bin
```

and typical use would be:

```
cprof -e -n your.exe vmon.out.n > cprofile_n.txt &
```

In order to get useful data from cprof, it helps if all sources are compiled with -g, and that any intermediate files used during compilation are saved in directories that are accessible when cprof is used, so that source-file annotation can be as complete as possible. An example of annotated source-code is shown below:

```
 -----------------------------------------------------------------------------
 line  tics       source-code
-----------------------------------------------------------------------------
  521                elseif(itype.eq.2) then
  522     6             do jlist=1,nlist
  523    22                if(and(jspinst(jlist),ispin(12)) /= 0 ) cycle
  524    56                c1=fprod*isipart44(jpartition)*cnphi(jlist)
  525     1                do ialpha=1,24
  526   137                   c2=c1*sgnalpha(ialpha)
  527    94                   isospin2=isoalpha(ialpha)+isopshell
  528    60                   ii=jspinst(jlist)+ispalpha(ialpha)
  529                         do ilist=1,nlist
  530                            cadd=c2*cpphi(ilist)
  531                            iispin=ii+ispinst(ilist)
  532 26916                      cphi(iispin,isospin2)=cphi(iispin,isospin2)+cadd
  533                         enddo
  534     9                enddo
  535     8             enddo
  536                endif
```

In this example, the vast majority of the clock-ticks hit on the main statement in the innermost loop. Since clock-ticks have a frequency of 100 per second, the data above shows about 269 seconds spent in this innermost loop. One should keep in mind that the optimizer mixes instructions over a range of source statements, and that interrupt sampling is quite coarse grained. So one should use judgment when interpreting the profiling data.

If you want to start profiling in MPI_Init() and stop profiling in MPI_Finalize() using the vmon.out format, it is not necessary to add calls to your code. You can simply set an env variable:

```
export VPROF_PROFILE=yes
```

and you should get vmon.out files written during the wrapper for MPI_Finalize(). Setting that env variable just calls vprof_start() in the wrapper for MPI_Init() and vprof_stop() in the wrapper for MPI_Finalize().

Since vprof/cprof uses the same mechanism and provides the same basic histogram data as traditional Unix-based profiling, it should be possible to use them interchangeably. At the present time, I have a primitive utility that can read a gmon.out file, extract the histogram data and write it in vmon.out format. That way you can use the -pg profiling method, optionally with moncontrol(), and still make use of the nice source-file annotation provided by cprof. This is quite experimental, but if you are really interested, you can check code in:

bgqssn4.watson.ibm.com:/bgsys/home2/walkup/codes/vprof/utils

There is a README and an example in that directory.

Hardware-counters.

Hardware counters for BGQ can provide very useful insight into application behavior, and some basic counter functionality has been built in to the MPI wrappers, using libraries:

libmpihpm.a : for pure MPI applications
libmpihpm_smp.a : for mixed OpenMP + MPI applications

When using these libraries, you also need to link with the basic counter libraries:

/bgsys/drivers/ppcfloor/bgpm/lib/libbgpm.a
/bgsys/drivers/ppcfloor/spi/lib/libSPI_upci_cnk.a

At the time of this writing, these libraries for BGQ use only the A2 counters, up to 24 counters per core, which are normally hardware-thread specific, and the L2 counters, which are shared across the node. The counters are accessed using the BGPM layer, which comes with some constraints. It is normally desirable to collect counter data on rather large blocks of code, and it is usually sufficient to look at the aggregate counts on a per-process basis. For threaded applications, this is best accomplished using the hardware-distributed mode provided by the BGPM layer. With this mode, one defines a mask that describes which hardware-threads "belong" to a given process, and those hardware-threads are attached to a "handle" that enables the master thread to read those counters. The choice of using hardware-distributed mode has the side-effect that counter aggregation is done by the BGPM layer … so it is not possible to see the counts for individual hardware threads (to do that would require use of the software-distributed mode of BGPM, which comes with other tradeoffs). For more information about the BGPM counter interface, see the documentation in :

/bgsys/drivers/ppcfloor/bgpm/docs/html

The current default settings for the libraries above are to use a predefined group of A2 counters, plus a set of L2 counters, and automatically provide aggregate counts on a per-node basis (not per-process).
The reason for looking at aggregate counts on a per-node basis is that the L2 counters are shared across the node, and so interpretation of those counters makes sense at the node level. The default set of A2 counters is:

```
// HPM_GROUP=0, a good basic set, can count 1-4 threads/core
static unsigned int mySet[6] = { PEVT_LSU_COMMIT_LD_MISSES,      //  0
                                 PEVT_LSU_COMMIT_CACHEABLE_LDS,  //  1
                                 PEVT_L1P_BAS_MISS,              //  2
                                 PEVT_INST_XU_ALL,               //  3
                                 PEVT_INST_QFPU_ALL,             //  4
                                 PEVT_INST_QFPU_FPGRP1};         //  5
```

With those counters one can get the instruction counts for the integer and floating-point pipelines, total floating-point operations, and useful information about L1 D-cache misses and misses for the prefetch buffer. When combined with data from the L2 counters, a useful picture can be obtained. By default, the hardware-counter outputs will include a section with both the raw counts, summed over processes on a given node, and some derived metrics.

When you use one of these libraries, you should get output files:

hpm_summary.#.rank

where the rank is the MPI rank in MPI_COMM_WORLD … the same output file controls are used for the mpi_profile.#.rank files … but the hpm summary files normally contain data summed over all processes that shared the node with the specified MPI rank. Example output is shown here:

```
======================================================================
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)
```

This particular case was from the Sequoia benchmark SPHOT, using 8 MPI ranks per node, and 8 OpenMP threads per process. The section that describes data source for loads needs some explanation. The counters provide the total number of loads and the loads that missed in the L1 D-cache; so one can get the fraction of loads that hit in L1 D-cache from that data. When a load misses in the L1 D-cache, the next place to look is the L1P prefetch buffer; if there is a miss at that level, the request goes to L2, and if there is an L2 miss, the request goes to DDR. So the section that describes load "hits" refers to demand loads. In contrast, the total DDR traffic includes all activity, which is often dominated by stream prefetching rather than demand loads. It may be that the fraction of loads that hit in DDR is quite small, and yet the total DDR traffic can be very large due to prefetch activity.

It is useful to examine the counter output as a function of the number of hardware threads used per core. One example is shown here: SPHOT, where the main performance problem is that the code has branches with fairly long sets of dependent calculations that cannot be pipelined effectively. The multiple hardware threads on BGQ provide an excellent way to increase the instruction throughput, and the cache behavior is very favorable:

```
SPHOT              threads/core              1        2        4
--------------------------------------------------------------------------
Instructions per cycle completed per core = 0.2283   0.4345   0.7079
Per cent of max issue rate per core =       22.83    29.53    48.14 %
Total weighted GFlops for this node =       3.520    6.895    11.503
Loads that hit in L1 d-cache =              92.18    92.14    88.92 %
                  L1P buffer =              0.55     0.22     0.22 %
                  L2 cache   =              7.28     7.64     10.86 %
                  DDR        =              0.00     0.00     0.00 %
Total DDR traffic (Bytes/cycle):    ld =    0.000    0.000    0.000
                                    st =    0.000    0.000    0.001
                                    total = 0.000    0.001    0.001
```

For SPHOT you can see that there is a very good speedup from 1 to 4 hardware threads per core … the GFlops increased by a factor of ~3.3. You can see that when going from 2 threads/core to 4 threads/core, there is some contention for L1 D-cache … there are fewer hits in L1 D-cache and more hits at the L2 level, which carries with it a much longer load latency. That is one of the reasons that the speed-up with 4 threads is not perfect. To obtain data like this, it is of course necessary to run multiple jobs, using 1-4 hardware threads per core. Overall for SPHOT at 4 hardware threads/core, each A2 core is cranking out instructions at about 48% of the theoretical limit … which is a good issue rate.

With the libmpihpm.a/libmpihpm_smp.a libraries, you should always get hardware counts for the code region from MPI_Init() to MPI_Finalize() … that code block is labeled "mpiAll" in the hpm_summary text files. You can optionally instrument your code with calls to HPM_Start/HPM_Stop to record counts for other code blocks. These calls should be made in a non-threaded section of code, or perhaps by just the master thread, due to use of the hardware-distributed mode of the BGPM layer. These calls take a character-string label:

C example:

```
    HPM_Start("timesteps");
    do_timesteps();
    HPM_Stop("timesteps");
```

Fortran example:

```
    call hpm_start('timesteps')
    call do_timesteps()
    cal hpm_stop('timesteps')
```

The HPM_Start/HPM_Stop calls can be nested, the only requirement is that for each call to HPM_Start, there should be one call to HPM_Stop with a matching label argument. Also … since the default method is to aggregate the counts over all processes that share a node, interpretation will be challenging (to say the least) unless all MPI ranks on a given node make the same sequence of HPM_Start/ HPM_Stop calls.

For C++ applications, these routines need to be declared as having extern "C" linkage:

```
extern "C" void HPM_Start(char *);
extern "C" void HPM_Stop(char *);
```

If you want to use other groups of A2 counters, you can specify some predefined groups by setting an environment variable. There are 24 hardware-counters per A2 core, and most are hardware-thread specific. If you want to use the same set of counters for all 4 hardware threads, then you are limited to six counters. In some cases you may want to count using just one hardware thread, in which case you can use up to 24 counters. For example, if you wanted to count all FPU-related instructions you could set an environment variable:

```
export HPM_GROUP=2
```

and the resulting hpm_summary.#.rank file should show just the counts for that process, with no derived metrics. The sets of counters may change over time … you can check the source file hpm.c in directory:

```
bgqssn4.watson.ibm.com:/bgsys/home2/walkup/mpi_trace/bgq/src
```

for the groups that are currently defined. The default group, HPM_GROUP=0 is sufficient for many purposes.

Event-tracing.

The idea for event tracing is to obtain insight into the time-dependent nature of messaging between various MPI processes. It is often possible to visually spot problems such as load imbalance, or coding issues that can result in effective serialization, or less than ideal parallelization in the code. The most common display is an x-y plot with time as the x-axis and MPI rank as the y-axis, using colored rectangles to represent each MPI event. There have been many similar tools in use for MPI applications for many years, such as jumpshot and vampire trace. A key problem has been that it is really easy to generate an unwieldy amount of data. The approach taken here is to be as selective as possible about event tracing, in order to keep the data volume manageable, and then use a lightweight viewer that can relate each MPI event back to source code. To activate event-tracing in your code, the preferred mechanism is to insert calls:

C example:

```
trace_start();  // start event-tracing
do_work();
trace_stop();  // stop tracing
```

Fortran example:

```
call trace_start()   ! start tracing
call do_work()
call trace_stop()  ! stop tracing
```

If you have an application that makes regular time-steps or iterations of some kind, it is usually sufficient to trace a few iterations or time-steps, because the pattern should repeat. Also, in most MPI applications, many of the MPI ranks are ostensibly doing the same kind of thing … so it is possible to get insight into the time-dependent behavior by looking at a subset of MPI processes.

Instead of instrumenting your code with trace_start()/trace_stop() calls, you can tell the MPI wrappers to start event tracing in the wrapper for MPI_Init(), by setting an environment variable:

```
export TRACE_ALL_EVENTS=yes
```

When event tracing is enabled, each call to an MPI function takes 48 bytes, and a small buffer is reserved in memory to hold these event records on each MPI rank. The default buffer size is enough to hold records for 50000 MPI calls, which takes 2.4*10^6 bytes of memory. Once the trace buffer is full, additional event records are discarded … so you can get a maximum of 50000 events saved in the trace buffer on each rank. If that is not sufficient, you can set the buffer size at run-time with an environment variable:

```
export TRACE_BUFFER_SIZE=4800000  (for example)
```

where the value is in bytes … the example above would be sufficient for 10^5 MPI calls per rank. It is best if the total volume of trace data can be kept to less than a few hundred MB, otherwise trace visualization will be unmanageable, so keep that limitation in mind when setting the buffer size. If the trace-buffer overflows, you will get a warning message when the application calls MPI_Finalize(). The trace-buffer limitation applies both to selective tracing using trace_start()/trace_stop() and to tracing that starts in MPI_Init() via the

TRACE_ALL_EVENTS environment variable.

Some applications make very frequent calls to routines such as MPI_Iprobe(), which is often called in a loop, waiting for a message to come in. Such frequently-called routines can quickly overflow any reasonable-sized trace buffer, so it may be necessary to disable event-tracing for such MPI routines. You can do this by setting an environment variable:

```
export TRACE_DISABLE_LIST=MPI_Iprobe,MPI_Comm_rank  (for example)
```

using a comma-separated list of MPI routines that you want to exclude from the event tracing … those routines will still be included in the overall timing summary files, mpi_profile.#.rank.

Since the graphical display is an x-y plot with MPI rank as the y-axis, it is most convenient to display data for MPI ranks ranging from 0 to some maximum value. Experience has been that it is often sufficient to examine ranks 0-255, so this is set as the default range. You can set an environment variable TRACE_MAX_RANK=N, which sets the limiting rank to N, that is you get trace data for ranks 0 through N-1 saved in the aggregate trace file, written when the application calls MPI_Finalize(). You could optionally set TRACE_ALL_TASKS=yes, if you really want to save events from all MPI ranks, but that would be asking for trouble (too much data) for very large-scale parallel jobs.

The output from event-tracing is a binary file "events.trc", which contains the concatenated records (48 bytes each) for all of the events saved, ordered by rank in MPI_COMM_WORLD. There is a primitive trace visualization tool, traceview, for display of this data. The traceview utility is written using OpenGL, and it is intended to be used locally on your laptop or workstation, because graphics-intensive applications work best that way. It is also possible to use a version built for a Linux front-end, and use X-windows to display the data. That would require an X-server on your local display with the OpenGL extensions … but it is highly recommended to use a local copy of traceview. This utility uses "glut" and "glui" software layers which are broadly available, and so traceview has been built on Windows, cygwin, Linux-x86, Linux-on-power, AIX, and Apple OS X.

Typical use of traceview would be:

```
traceview events.trc
```

There is a help button which describes most of the things that you need to know, including the key assignments for controlling the viewing region. Some example screen shots are shown below.

In this display the x-axis is elapsed time in the job, and the y-axis is MPI rank from 0-31, taken from a 2048-way parallel job using the MILC NSF benchmark. MILC uses a conjugate-gradient solver, and the code was instrumented with calls to trace_start()/trace_stop() in order to capture a few iterations. The colored bars correspond to MPI routines : orange is MPI_Wait, green is MPI_Allreduce. This data was recorded on BlueGene/P during an early stage of system-software development. The relatively long times that are sometimes spent in MPI_Wait() were due to errors in the messaging layers for BGP … the messaging software was not properly handling multiple outstanding non-blocking calls to MPI_Isend()/MPI_Irecv(). In this example, the code was attempting to overlap computation and communication by using a sequence of steps like this:

```
call MPI_Isend/MPI_Irecv for the first set of messages
do some work
call MPI_Isend/MPI_Irecv for a second set of messages
call MPI_Wait for the first set of messages
do more work
call MPI_Wait for the second set of messages
```

Unfortunately, some of the MPI_Wait() calls for the first set of messages did not return until some of the MPI_Wait() calls for the second set of messages completed. The result was poor parallel performance … some MPI ranks were stuck waiting for others to finish computational steps before they could proceed. In this case the defect was reported, but it was possible to re-structure the code to avoid the problem:

```
do some work
call MPI_Isend/MPI_Irecv/MPI_Wait for the first set of messages
do more work
call MPI_Isend/MPI_Irecv/MPI_Wait for the second set of messages
```

This second approach makes no attempt to overlap computation and communication, but it did avoid the problem with message-completion. A display of the MPI trace file for the modified code is shown below.

Black corresponds to computation, and one can see that now all of the MPI ranks are in sync, and the MPI events take a very small fraction of the elapsed time. The image above is a good example of a well-behaved parallel application … all of the MPI ranks are busy working concurrently, the load is beutifully balanced … there are no ranks unduly waiting on others to finish some stage of the work, and the fraction of time spent in MPI is small.

A key feature of the trace viewer is the ability to map MPI events back to the source-code location. Each trace record includes the instruction-address for the MPI routine and the grandparent, going up the call-stack. When you click (left-mouse button) on an MPI event, the details for that event will be displayed, and then you can use the addr2line utility to translate from instruction address to source-file and line-number. This requires -g as one of your options for compilation and linking. Example output from clicking on an MPI event is shown here:

```
task id = 62, event = MPI_Barrier
  tbeg = 45.193914, tend = 45.229736, duration = 35.823 msec
  parent address = 0x0115648c
  grandparent address = 0x01139150
```

In this example, the MPI routine was MPI_Barrier, and you can find the source-file and line-number from the instruction addresses, using the addr2line utility. The event records include destination-ranks for flavors of MPI_Send, source ranks for flavors of MPI_Recv, and message-sizes where that information is available. The key information, however, is the instruction-address.

An example of an application with inherent load imbalance is shown below:

In this example the time spent in MPI_Barrier (the yellow rectangles) increases roughly linearly with MPI rank, and a similar pattern occurs for MPI_Bcast (light orange rectangles). In this application (the GFS code from the US National Weather Service), the load imbalance arises naturally from the parallel decomposition strategy. At this scale, load imbalance, not network latency or bandwidth, is what limits the parallel efficiency.

Sometimes applications have their own messaging layers … for example MPI_Isend() might always be called from an application routine “my_send()”. In cases like that you may need to look at the grandparent address, or even deeper into the call-stack for deeply layered cases. You can save instruction address data starting at any point in the call stack, by setting the environment variable TRACEBACK_LEVEL to the appropriate value … but you have to set that when you record the trace data because each event record has space for just two entries (called parent and grandparent) for instruction addresses.

In some cases it would be better to display performance metrics in a format that reflects the physical problem for the simulation. One example is shown below, from a cubed-sphere model of the earth's atmosphere:

The data in this figure is the total amount of time spent in MPI routines, for each of 31104 MPI ranks, from a BlueGene/P job. Dark blue corresponds to the smallest time spent in MPI, and red corresponds to the largest time spent in MPI. The data looks like a gift-wrapped planet earth, and the timing variations are all due to load imbalance. The code uses a 2D decomposition for each of the six faces of the cube. In this case there were 72x72 MPI ranks for each face, and a total of 2000 grid points in each of the two dimensions. The yellow stripe occurs because the number of grid points (2000) is not evenly divisible by 72 … the MPI ranks in the yellow stripe have one less grid point. As a result, they finish their computation sooner, and wait in MPI longer. The MPI ranks in the red square have one fewer grid point in each of two dimensions, so they do the least amount of work, and wait in MPI the longest. Finally, MPI ranks that are positioned on a land mass have some extra work to do, relative to ranks positioned over the ocean … so ranks over the ocean must wait for the ones over land to finish their extra work … ranks over land have the most work, and do the smallest amount of waiting in MPI.

It would be possible to use similar kinds of displays for time-dependent data, such as event tracing, but this kind of approach is clearly dependent on the nature of the simulation, and so there is not much in the way of general tools that map performance data back to the physical simulation domain.