The IBM High Performance Computing Toolkit on BlueGene/L

Kirk E Jordan, Ph.D.
Strategic Growth Business/Deep Computing
Systems & Technology Group
kjordan@us.ibm.com

Collaborators:
I-Hsin Chung - ACTC
Bob Walkup – Physical Sciences
Outline

- Performance Decision tree
- IBM High Performance Computing Toolkit
  - MPI performance: MP_Profiler
  - CPU performance: Xprofiler, HPM
  - Modular I/O: MIO
  - Visualization and analysis: PeekPerf

- Challenges
- Future Work
Performance Decision Tree

Total Performance

Computation
- Xprofiler
  - Routines/Source
- HPM
  - Summary/Blocks
- Compiler
  - Source Listing

Communication
- MP_Profiler
  - Summary/Events

I/O
- MIO Library
IBM High Performance Computing Toolkit on BG/L

- MPI performance: MP_Profiler, MP_Tracer
- CPU performance: Xprofiler, HPM
- Visualization and analysis: PeekPerf
- Modular I/O: MIO
Message-Passing Performance:

- **MP_Profiler Library**
  - Captures “summary” data for MPI calls
  - Source code traceback
  - User **MUST** call MPI_Finalize() in order to get output files.
  - No changes to source code
    - MUST compile with –g to obtain source line number information

- **MP_Tracer Library**
  - Captures “timestamped” data for MPI calls
  - Source traceback
Compiling and Linking Example

BGL=/bgl/BlueLight/ppcfloor
CC=$(BGL)/blrts-gnu/powerpc-bgl-blrts-gnu/bin/gcc
CFLAG= -I $(BGL)/bglsys/include
MPI_LIB= -L $(BGL)/bglsys/lib -lmpich.rts -lmsglayer.rts -lirts.rts -ldevices.rts
TRACE_LIB= -L $(MP_PROFILER) -lmpitrace.rts
BINUTILS_LIB= -L $(BINUTILS) -lbfd -liberty

target: source.c
  $(CC) –o $@ $< $(CFLAG) $(TRACE_LIB) $(MPI_LIB) $(BINUTIL_LIB)

$(TRACE_LIB) has to precede $(MPI_LIB)
### MP_Profiler Summary Output

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>3</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>12994</td>
<td>0.0</td>
<td>0.016</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>19575</td>
<td>11166.9</td>
<td>13.490</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>910791</td>
<td>5804.2</td>
<td>9.216</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>138173</td>
<td>2767.9</td>
<td>73.835</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>784936</td>
<td>15891.6</td>
<td>2.407</td>
</tr>
<tr>
<td>MPI_Sendrecv</td>
<td>894809</td>
<td>352.0</td>
<td>88.705</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>1537375</td>
<td>0.0</td>
<td>288.049</td>
</tr>
<tr>
<td>MPI_Waitall</td>
<td>44042</td>
<td>0.0</td>
<td>25.312</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>464</td>
<td>41936.8</td>
<td>3.272</td>
</tr>
<tr>
<td>MPI_BARRIER</td>
<td>1312</td>
<td>0.0</td>
<td>34.206</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>68</td>
<td>16399.1</td>
<td>2.680</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>6</td>
<td>17237.3</td>
<td>0.532</td>
</tr>
</tbody>
</table>

Total communication time = 770.424 seconds.
Total elapsed time = 1168.662 seconds.
User cpu time = 1160.960 seconds.
System time = 0.620 seconds.
Maximum memory size = 68364 KBytes.

To check load balance: grep "total comm" mpi_profile.*
MP_Profiler Sample Call Graph Output

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>2311</td>
<td>143.734</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>2311</td>
<td>0.206</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td></td>
<td></td>
</tr>
<tr>
<td>communication time = 143.940 sec, parent = gwwrloc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>91959</td>
<td>137.823</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>communication time = 137.823 sec, parent = f2drecv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>23607</td>
<td>106.821</td>
</tr>
<tr>
<td>MPI_Gatherv</td>
<td>23607</td>
<td>2.139</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td></td>
<td></td>
</tr>
<tr>
<td>communication time = 108.960 sec, parent = puttsf</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>6378</td>
<td>94.435</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>communication time = 94.435 sec, parent = fft_tri_recv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Routine</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>93003</td>
<td>83.836</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td></td>
<td></td>
</tr>
<tr>
<td>communication time = 83.836 sec, parent = fft2drecv</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
IBM Systems & Technology Group

Deep Computing

MP_Profiler Output with Peekperf

```
return
end

subroutine rcv_real(orig, value, size, tag, message)
imPLICIT none
```
Measuring Communication Performance

The MPI standard provides alternate entry points for timing:

```c
MPI_Send(...) {
    Start_timer();
    PMPI_Send(...);
    Stop_timer();
    Log_the_event();
}
```

Implement a scalable approach that can support >10**5 MPI tasks.

Two modes: summary and event-logging.
MP_Profiler: Summary Mode

- Low overhead: <0.1 microsecond per call
- Low data volume: save just a few small text files
- Make use of the parallelism to do data collection and analysis
  - Save details for MPI ranks with minimum, maximum, and median communication times.
  - Collect total communication time for all ranks in one text file.
  - Collect data for all hardware counter events in one shot.
  - Automatically filter other profile data from gprof etc.
MPI Timing Summary

MPI Rank 0 out of 1024.
-----------------------------------------------------------------
elapsed time from clock-cycles using freq = 700.0 MHz.
-----------------------------------------------------------------
total communication time = 4.943 seconds.
total elapsed time = 43.113 seconds.
top of the heap address = 158.305 MBytes.
-----------------------------------------------------------------

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>2</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>2</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>171</td>
<td>765163.0</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>174</td>
<td>757120.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>340</td>
<td>0.0</td>
<td>1.912</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>11</td>
<td>0.0</td>
<td>0.038</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>11</td>
<td>16.7</td>
<td>2.992</td>
</tr>
</tbody>
</table>
MP_Profiler : Event Logging

- Visualize the time-sequence of MPI events.
- Preserve the connection between MPI events and source code.
- Use a lightweight fast viewer based on OpenGL.
- Keep the data volume manageable.
- Limit event logging to a subset of MPI ranks, and/or to a subset of the overall simulation.
MPI Events for Enzo - Original

```
task id = 23, event = MPI_Wait
tbeg = 693.203052, tend = 713.751245, duration = 20548.194 msec
parent address = 0x0011b210
grandparent address = 0x00122724
source = 19 (valid only for receive requests)
```
Enzo: Original Code

For each pair of regions
  conditionally call MPI_Test(…) to check for outstanding send requests
  if regions overlap
    call MPI_Isend(send_request,…) to “send” data

For each pair of regions
  if regions overlap
    call MPI_Irecv(recv_request) to receive boundary data
    call MPI_Wait(recv_request,…)
Enzo : Modified Code

For each pair of regions
  conditionally call MPI_Test(…) to check for outstanding send requests
  if regions overlap
    call MPI_Isend(send_request,…) to “send” data

call MPI_Barrier(…)

For each pair of regions
  if regions overlap
    call MPI_Irecv(recv_request) to receive boundary data
    call MPI_Wait(recv_request,…)
MPI Events for Enzo - Tuned
Xprofiler

- CPU profiling tool similar to gprof
- Can be used to profile both serial and parallel applications
- Use procedure-profiling information to construct a graphical display of the functions within an application
- Provide quick access to the profiled data and helps users identify functions that are the most CPU-intensive
- Based on sampling (support from both compiler and kernel)
- Charge execution time to source lines and show disassembly code
Running Xprofiler

- Compile the program with -pg
- Run the program
- gmon.out file is generated (MPI applications generate gmon.out.1, …, gmon.out.n)
- Run Xprofiler
Xprofiler - Application View

Program: sim0
Total CPU Usage: 1,31 seconds (summary of 1 gmon.out profile files)
Display Status: showing 11 out of 95 nodes and 9 out of 96 arcs
Xprofiler: Main Display

- **Width of a bar:** time including called routines
- **Height of a bar:** time excluding called routines
- **Call arrows labeled with number of calls**
- **Overview window for easy navigation** (View → Overview)
Xprofiler: Flat Profile

- **Menu Report** provides usual gprof reports plus some extra ones
  - Flat Profile
  - Call Graph Profile
  - Function Index
  - Function Call Summary
  - Library Statistics

![Screenshot of Xprofiler Flat Profile](image)
Xprofiler: Source Code Window

- Source code window displays source code with time profile (in ticks=.01 sec)

- Access
  - Select function in main display
  - ➔ context menu
  - Select function in flat profile
  - ➔ Code Display
  - ➔ Show Source Code

```c
217 229  t21 = t21 + [(i+1)*n+k]*bt[j*n+k];
```

```c
202 */ use 2x-unrolling of the outer two loops */
203 /*
204 for (i=0; i<10*i0+1; i++)
205 {
206   for (j=0; j<10*j0+1; j++)
207   {
208     t11 = c[i0+n0];
209     t12 = c[i0+n1];
210     t21 = c[i0+n0+j0];
211     t22 = c[i0+n1+j1];
212     for (k=0; k<k0; k++)
213       t21 = t21 + a[0+(i+1)*n+k]*bt[j*n+k];
214   }
215 }
216 144 t22 = t22 + a[i0+n0+k]*bt[(j+1)*n+k];
217 219 c[i0+n0] = t11;
220 c[i0+n1] = t12;
221 c[i0+n0+j0] = t21;
222 c[i0+n1+j1] = t22;
223 5 for (j=0; j<j0; j++)
224   { t11 = c[i0+n0+j];
225    t21 = c[i0+n0+j0];
226    for (k=0; k<k0; k++)
227      { t11 = t11 + a[0+n0+j]*bt[j*n+k];
228       t21 = t21 + a[0+n0+k]*bt[j*n+k];
229     }
230   }
231 232 c[i0+n0] = t11;
233 c[i0+n0+j0] = t21;
234 235 236 }
237 )
```
Xprofiler - Disassembler Code

<table>
<thead>
<tr>
<th>address</th>
<th>no. ticks</th>
<th>instruction</th>
<th>assembler code</th>
<th>source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>10002E13</td>
<td>81</td>
<td>fcc4287c</td>
<td>fms</td>
<td>6,4,1,5</td>
</tr>
<tr>
<td>10002E1C</td>
<td>64</td>
<td>ccf70008</td>
<td>lfd</td>
<td>7,0x8(23)</td>
</tr>
<tr>
<td>10002E20</td>
<td>187</td>
<td>c90c0008</td>
<td>lfd</td>
<td>8,0x8(12)</td>
</tr>
<tr>
<td>10002E24</td>
<td>53</td>
<td>c9750008</td>
<td>lfd</td>
<td>11,0x8(21)</td>
</tr>
<tr>
<td>10002E28</td>
<td>89</td>
<td>fd63582a</td>
<td>fa</td>
<td>11,3,11</td>
</tr>
<tr>
<td>10002E2C</td>
<td>63</td>
<td>fd28387c</td>
<td>fms</td>
<td>9,8,1,7</td>
</tr>
<tr>
<td>10002E30</td>
<td>4</td>
<td>dd5b0008</td>
<td>stdu</td>
<td>10,0x8(27)</td>
</tr>
<tr>
<td>10002E34</td>
<td>113</td>
<td>fcca302a</td>
<td>fa</td>
<td>6,10,6</td>
</tr>
<tr>
<td>10002E3C</td>
<td>27</td>
<td>c8760008</td>
<td>lfd</td>
<td>3,0x8(22)</td>
</tr>
<tr>
<td>10002E40</td>
<td>87</td>
<td>fd8012fa</td>
<td>fma</td>
<td>12,0,11,2</td>
</tr>
<tr>
<td>10002E44</td>
<td>35</td>
<td>dc850008</td>
<td>stdu</td>
<td>5,0x8(25)</td>
</tr>
<tr>
<td>10002E48</td>
<td>4</td>
<td>fd63482a</td>
<td>fa</td>
<td>3,3,9</td>
</tr>
<tr>
<td>10002E4C</td>
<td>12</td>
<td>cd5a0008</td>
<td>lfd</td>
<td>10,0x8(26)</td>
</tr>
<tr>
<td>10002E50</td>
<td>62</td>
<td>fcc021ba</td>
<td>fma</td>
<td>6,0,6,4</td>
</tr>
<tr>
<td>10002E54</td>
<td>36</td>
<td>c85b0008</td>
<td>lfd</td>
<td>2,0x8(27)</td>
</tr>
<tr>
<td>10002E58</td>
<td>244</td>
<td>dce0c0008</td>
<td>stdu</td>
<td>7,0x8(12)</td>
</tr>
<tr>
<td>10002E5C</td>
<td>28</td>
<td>fd0040fa</td>
<td>fma</td>
<td>8,0,3,8</td>
</tr>
<tr>
<td>10002E60</td>
<td>4</td>
<td>c8990008</td>
<td>lfd</td>
<td>4,0x8(25)</td>
</tr>
<tr>
<td>10002E64</td>
<td>316</td>
<td>dcd00008</td>
<td>stdu</td>
<td>6,0x8(20)</td>
</tr>
<tr>
<td>10002E68</td>
<td>29</td>
<td>fcc62507c</td>
<td>fms</td>
<td>3,2,1,10</td>
</tr>
</tbody>
</table>

POLD(I,J) = P(I,J) + ALPHA*(PHEN(I,J) - U(I,J)) + ALPHAPU*U(I,J)
Hardware Performance Monitor (HPM)

- Provides comprehensive reports of events that are critical to performance on IBM systems.
- Gather critical hardware performance metrics, e.g.
  - 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.
LIBHPM

- Instrumentation library
- Provides performance information for instrumented program sections
- Supports multiple (nested) instrumentation sections
- Multiple sections may have the same ID
- Run-time performance information collection
- Based on bgl_perfctr layer – can be eliminated in BG/P
Event Sets

- 16 sets (0-15); 328 events
- Information for
  - Time
  - FPU (0,1)
  - L3 memory
  - Processing Unit (0,1)
  - Tree network
  - Torus network
- For detailed names and descriptions: event_sets.txt
Functions

- **hpmlInit( taskID, progName ) / f_hpminit( taskID, progName )**
  - taskID is an integer value indicating the node ID.
  - progName is a string with the program name.

- **hpmsStart( instID, label ) / f_hpmstart( instID, label )**
  - instID is the instrumented section ID. It should be > 0 and <= 100 (can be overridden)
  - Label is a string containing a label, which is displayed by PeekPerf

- **hpmsStop( instID ) / f_hpmstop( instID )**
  - For each call to hpmsStart, there should be a corresponding call to hpmsStop with matching instID

- **hpmsTerminate( taskID ) / f_hpmterminate( taskID )**
  - This function will generate the output. If the program exits without calling hpmsTerminate, no performance information will be generated.
Functions (continued)

- `hpmGetTimeAndCounters( numCounters, time, values )` / `f_GetTimeAndCounters ( numCounters, time, values )`
  - returns the time in seconds and counts since the call to `hpmInit`.
  - `numCounters`: integer indicating the number of counters to be accessed.
  - `time`: double precision float
  - `values`: “long long” vector of size “numCounters”.

- `hpmGetCounters( values )` / `f_hpmGetCounters ( values )`
  - Similar to `hpmGetTimeAndCounters`
  - only returns the total counts since the call to `hpmInit`
Example of Instrumented Code

- **C / C++**
  
  **Declaration:**
  
  ```c
  #include "libhpm.h"
  ```

  **Use:**
  
  ```c
  hpmInit( taskID, "my program" );
  hpmStart( 1, "outer call" );
  do_work();
  hpmStart( 2, "computing meaning of life" );
  do_more_work();
  hpmStop( 2 );
  hpmStop( 1 );
  hpmTerminate( taskID );
  ```

- **Fortran**
  
  **Declaration:**
  
  ```fortran
  include "f_hpm.h"
  ```

  **Use:**
  
  ```fortran
  call f_hpminit( taskID, "my program" )
  call f_hpmstart( 1, "Do Loop" )
  do …
      call do_work()
      call f_hpmstart( 5, "computing meaning of life" );
      call do_more_work();
      call f_hpmstop( 5 );
  end do
  call f_hpmstop( 1 )
  call f_hpmterminate( taskID )
  ```
Hpmcount Example

bash-2.05a$ hpm_count swim
…… // program output
hpmcount (V 2.5.4) summary
Execution time (wall clock time) : 7.378159 seconds
######## Resource Usage Statistics ########
Total amount of time in user mode : 0.010000 seconds
Average shared memory use in text segment : 672 Kbytes*sec

........
PM_FPU_FDIV (FPU executed FDIV instruction) : 0
PM_FPU_FMA (FPU executed multiply-add instruction) : 0
PM_CYC (Processor cycles) : 54331072
PM_FPU_STF (FPU executed store instruction) : 2172
PM_INST_CMPL (Instructions completed) : 17928229
Utilization rate : 0.446 %
Total load and store operations : 0.004 M
MIPS : 2.140
Instructions per cycle : 0.330
HPM Visualization Using PeekPerf
Environment Flags

- **HPM_EVENT_SET**
  - Select the event set to be recorded
  - Integer (0 – 15)

- **HPM_NUM_INST_PTS**
  - Overwrite the default of 100 instrumentation sections in the app.
  - Integer value > 0

- **HPM_WITH_MEASUREMENT_ERROR**
  - Deactivate the procedure that removes measurement errors.
  - True or False (0 or 1).

- **HPM_OUTPUT_NAME**
  - Define an output file name different from the default.
  - String

- **HPM_VIZ_OUTPUT**
  - Indicate if “.viz” file (for input to PeekPerf) should be generated or not.
  - True or False (0 or 1).

- **HPM_TABLE_OUTPUT**
  - Indicate table text file should be generated or not.
  - True or False (0 or 1).
Modular I/O (MIO)

- Addresses the need of application-level optimization for I/O.
- Analyze and tune I/O at the application level
  - For example, when an application exhibits the I/O pattern of sequential reading of large files
  - MIO
    - Detects the behavior
    - Invokes its asynchronous prefetching module to prefetch user data.
- Planned Integration into HPC Toolkit with PeekPerf capabilities
  - Source code traceback
  - Future capability for dynamic I/O instrumentation
Challenges

- **Hardware**
  - Different performance counter set

- **Scalability**
  - 64k nodes
  - Tracking communications for each pair:
    - $$(65,536)^2 \times 48 \text{ bytes} = 200 \text{ GB}$$
  - Number of processes and events vs. number of screen pixels
Future work

- **MP_Profiler**
  - Scalability
- **Xprofiler**
  - Improved GUI
  - Integration with other platforms
- **HPM**
  - Useful derived metrics and verifications
  - Hpmcount / Hpmstat ?
  - Integration with other platforms

- **Next target**
  - Modular I/O: MIO